6800 PROGRAMMING
FOR LOGIC DESIGN

BY ADAM OSBORNE

6800 PROGRAMMING
FOR LOGIC DESIGN

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

QUICK INDEX

QUICK INDEX (Continued)

QUICK INDEX (Continued)

6800 PROGRAMMING
FOR LOGIC DESIGN

Chapter 1
INTRODUCTION

This book explains how an assembly language program within a microcomputer system can replace combinatoriel logic — that is, the combined use of "off-the-shelf", non-programmable logic devices, such as standard 7400 series digital logic.

COMBINATORIAL LOGIC

If you are a logic designer, this book will teach you how to do your old job in a new way — by creating assembly language programs within a microcomputer system.

If you are a programmer, this book will show you how programming has found a new purpose — in logic design.

This is a "how to do it" book; as such, it has to become very specific, so a particular type of microcomputer, the MC6800, is referenced directly.

Companies manufacturing these microcomputers are:

MOTOROLA, INCORPORATED
Semiconductor Products Division
3501 Ed Bluestein Boulevard
Austin, Texas 78721

AMERICAN MICROSYSTEMS
3800 Homestead Road
Santa Clara, California 95051

WHAT THIS BOOK ASSUMES YOU KNOW

This book is a sequel to "An Introduction To Microcomputers", which was a single volume in its first edition, but is two volumes in its second edition.

"An Introduction To Microcomputers" describes microprocessors and microcomputers conceptually; it does not address itself to the practical matter of implementing a concept. This book addresses the practical matter of implementation.

In that this book is a sequel, it makes a single assumption — that you have read, or you otherwise understand the material covered in "An Introduction To Microcomputers". However, before launching into a real design project, you will need vendor literature that specifically describes the devices you have elected to use.

Note in particular that hardware and timing are not described in this book, either for the MC6800 CPU, or any other microcomputer devices; sufficient information may be found in "An Introduction To Microcomputers", Volume II — Some Real Products.

The MC6800 instruction set is described in Chapter 6 of this book, since programming is what this book is all about.

UNDERSTANDING ASSEMBLY LANGUAGE

Assembly lenguage instructions are the transfer functions of a microcomputer system; taken together, they constitute an "instruction set", which describes the individual operations which the microcomputer can perform.

You define the events which must occur within the microcomputer system serially — as a sequence of instructions, which, taken together, constitute an assembly language program.

In reality, understanding what individual instructions do within a microcomputer system is very straightforward; it is one of the simplest aspects of working with microcomputers. Yet it unduly terrifies users who are new to programming. If that includes you, a word of advice — forget about mnemonics and instruction sets; take instructions one at a time as you encounter them in this book. When you do not understand what an instruction is doing, look it up in Chapter 6.

The specter of "programming" will haunt you only if you let it.

HOW THIS BOOK HAS BEEN PRINTED

Notice that text in this book has been printed in boldface type and lightface type. This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type. Therefore, only read boldface type until you reach a subject about which you want to know more, at which point start reading the lightface type.

Chapter 2
ASSEMBLY LANGUAGE AND
DIGITAL LOGIC

THE DESIGN CYCLE

Any product that is to be built out of discrete digital logic components will go through a well defined design cycle.

```
DIGITAL
LOGIC
DESIGN
CYCLE
```

Let us assume that the product has been defined — from marketing management's point of view.

You are presented with a product specification which identifies necessary product performance and characteristics; your job is to deliver a viable design to manufacturing.
The design cycle will proceed as follows:
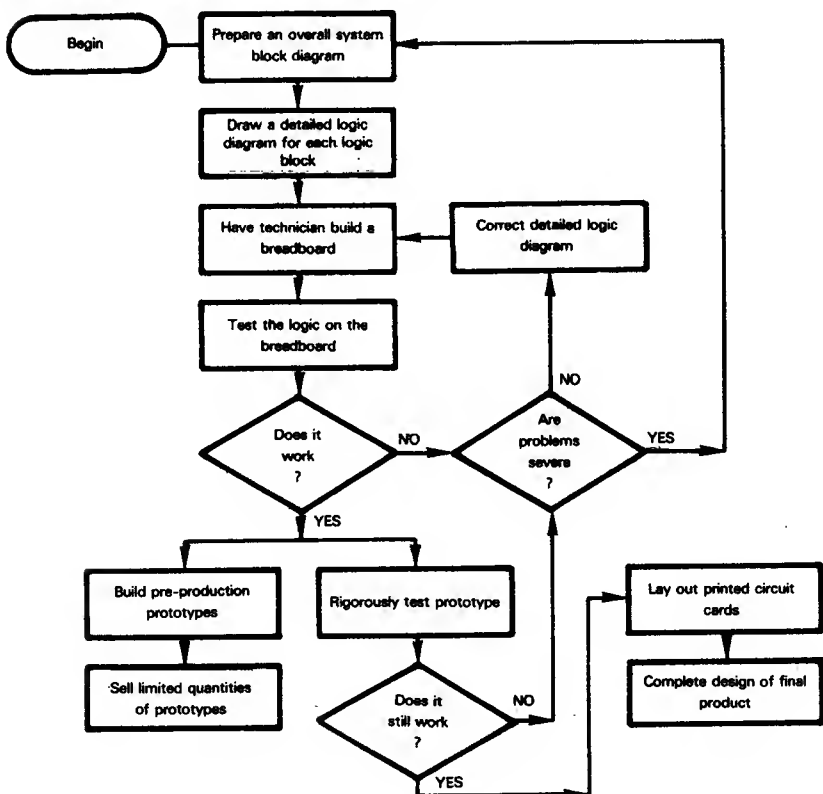
There is an expensive and slow iterative loop in any digital logic design cycle; as illustrated above, it consists of these steps:

- Redraw logic
- Build a new breadboard
- Test the breadboard for logic errors, technician errors or faulty components

This iterative loop makes combinatorial logic design slow and expensive — not only during the initial design phase, but even more so when you subsequently decide to modify or enhance the product.

What happens when you start using microcomputers? First of all, a portion of your logic vanishes into a "black box" — which is the microcomputer system:

> MICROCOMPUTER
> LOGIC DESIGN
> CYCLE

Your first step:

Prepare an overall
system block diagram
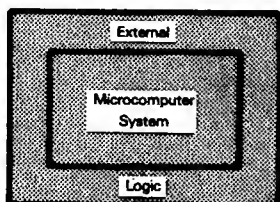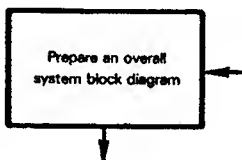
must now be broken out as follows:

| For microcomputer system, select device configuration | Divide logic into microcomputer system and external logic | For external logic prepare an overall logic block diagram |
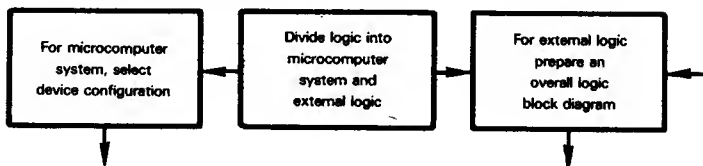
Partitioning your application into a microcomputer system and external digital logic may look like a difficult proposition — if you do not understand what the microcomputer system can do.

In fact, **once you have a microcomputer in your product, economics overwhelmingly favor making the "black box" assume as many tasks as possible; you must justify the existence of every single external logic gate.**

Remember, memory comes in finite increments. In order to expand the logic implemented within the microcomputer system, you may simply have to write additional instruction sequences that will reside in memory which would otherwise be wasted; adding program memory, for that matter, costs very little.

Also, compared to the cost of digital logic development, microcomputer logic development is quick and inexpensive. **A typical microcomputer system development cycle may be illustrated as follows:**
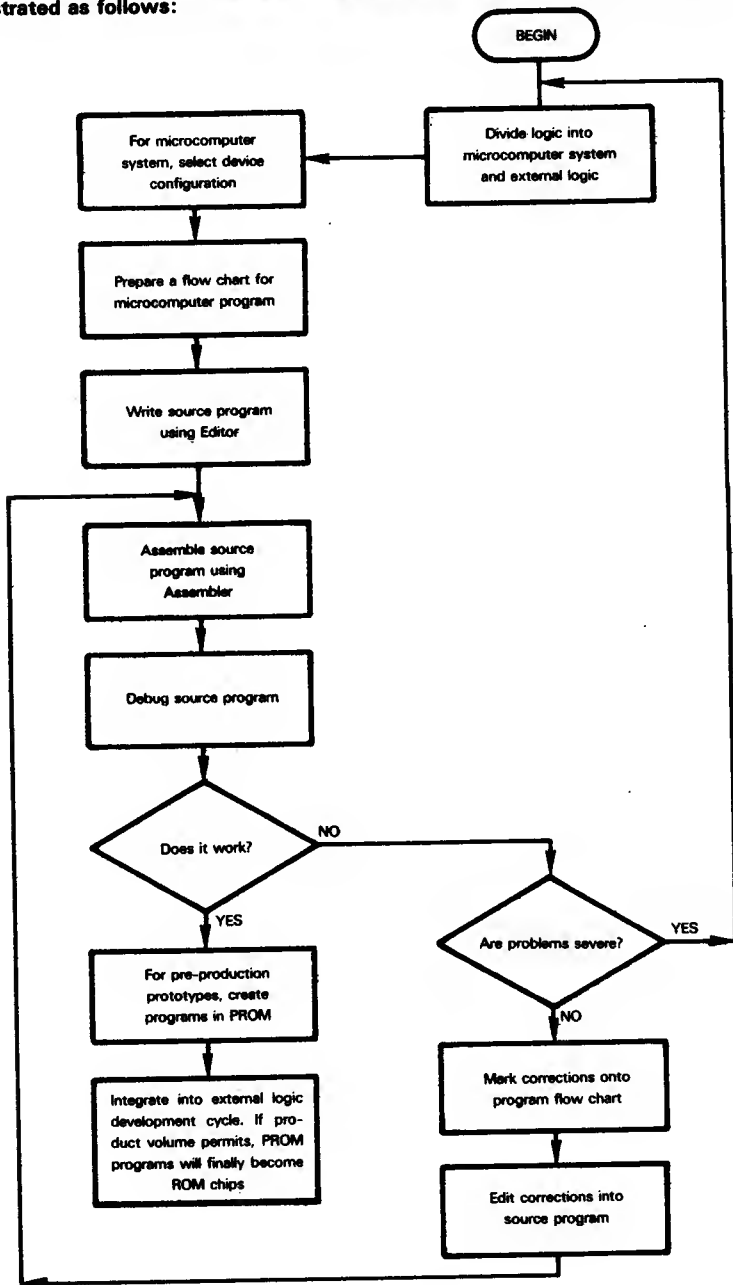
There are still iterative loops in the microcomputer development cycle illustrated above, but compared to digital logic development, less time and expense are associated with microcomputer development cycle iterative loops.

Every microcomputer is supported by a development system. Characteristics and operation of these development systems vary markedly from one company to the next; however they **all have these capabilities:**

1) You can **simulate the microcomputer system** you have configured without necessarily creating a breadboard.

2) You can execute a resident editor program to **create your source program.** Remember, a sequence of assembly language instructions is referred to as a "Source Program".

```
SOURCE
PROGRAM
```

3) You can **assemble the source program** right at the development system to create an object program. Remember, the source program becomes a sequence of binary digits, referred to as an object program, before it can be executed.

```
OBJECT
PROGRAM
```

4) You can **conditionally execute the object program** to make sure that it works.

Using a typical microcomputer development system, you can go through several major development cycles in a single day, where each development cycle might have taken one or two weeks in a total digital logic implementation. Within a single development cycle you can make many program corrections; in less than a minute you can make a simple correction, equivalent to adding or removing a gate (or MSI function) from a digital logic breadboard.

SIMULATING DIGITAL LOGIC

OK, so logic must eventually be separated into a microcomputer system, and logic beyond the microcomputer system.

We are going to have to address two aspects of this logic separation:

1) Based on the ability of assembly language to simulate digital logic, **we must develop some simple criterion for estimating what a microcomputer system can do** and what it cannot do.

2) **We must create a program to implement the logic functions which have been assigned to the microcomputer system.** Unfortunately, there are innumerable ways of writing a microcomputer program. Once you have mastered the concept of using instructions to drive a microcomputer system, **the next step is to learn how to write efficient programs.**

We will begin by describing simple digital logic simulation. This is a necessary beginning because there are some fundamental conceptual differences between digital logic and microcomputer programming logic.

MICROCOMPUTER SIMULATION OF A SIGNAL INVERTER

Suppose you want to invert a single signal:

```
1                               0
or ────────▷○──────── or
0                               1

        A = Ā
```
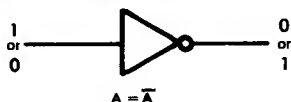
In the interests of developing good habits from the start, we will illustrate the signal inverter with the following logic flowchart:

FLOW CHART

```
┌─────────────────┐
│   Input signal  │
│  to be inverted │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│   Invert signal │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│  Output inverted│
│      signal     │
└─────────────────┘
```
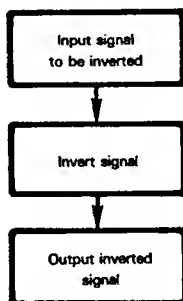
Although you would never use a microcomputer simply to replace a signal inverter, it is still worthwhile examining how it could be done.

A MICROCOMPUTER EVENT SEQUENCE

Recall that MC6800 type microcomputers have the following CPU registers:

CPU REGISTERS

```
┌──────────────────┐
│                  │   Status Register (S)
│                  │   Accumulator (A)
│                  │   Accumulator (B)
│                  │   Index Register (X)
│                  │   Program Counter (PC)
│                  │   Stack Pointer (SP)
└──────────────────┘
```
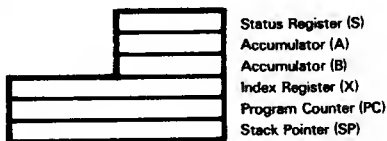
This single instruction:

      COM A   COMPLEMENT ACCUMULATOR A

when converted into object code and executed, inverts all eight bits of Accumulator A. But that does not duplicate the inverter. First, one binary digit of Accumulator A must be selected to represent the signal being inverted. But which one?

BIT DATA

Having decided which binary digit, how does it reach the Accumulator in the first place? And once inverted, how does the inverted bit become a signal again?

If the COM A instruction object code must be executed in order to perform the actual inversion, how and when does the object code reach the CPU? Clearly execution of this instruction must be timed to occur after the binary digit to be inverted has reached the Accumulator.

| DATA SOURCE AND DESTINATION |
| PROGRAM TIMING |

Steps needed to implement an inverter using a microcomputer may be illustrated by expanding our flowchart as follows:

Input
Signal

↓

┌─────────────────┐
│   Convert to   │
│ binary digit   │
└─────────────────┘

↓   Data/Signal
                     Source
┌─────────────────┐ Determination
│   Load into   │
│ Accumulator A │
└─────────────────┘

- - - - - - - - - - - - - - -

┌─────────────────┐
│ Load and execute │ Transfer
│ COM instruction │ Function
└─────────────────┘

- - - - - - - - - - - - - - -

┌─────────────────┐
│   Output from   │
│ Accumulator A │
└─────────────────┘

                     Data/Signal
                     Destination
┌─────────────────┐ Determination
│   Convert to   │
│ logic signal   │
└─────────────────┘

↓

Output
Signal
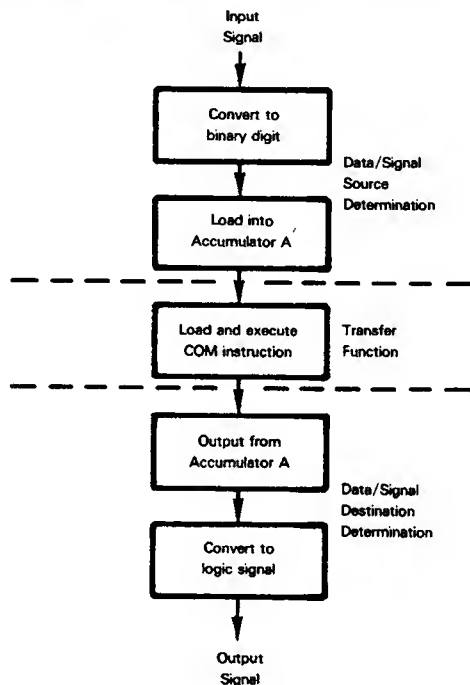
In the illustration above, pay most attention to the division of the problem into these three phases:

1) **Data/signal source determination.** We identify the data which is to be operated on. This data is transferred to a location out of which it can be accessed by the microcomputer Central Processing Unit (CPU).

2) **Transfer function execution.** The actual operation which must be performed on the source data will be referred to as a "Transfer Function".

3) **Data/signal destination determination.** The data or signals having been subject to the transfer function, must now be transferred to some destination.

We will now generate an instruction sequence to implement the three phases of the inverter simulation illustrated above.

Now one important point must be made regarding the microcomputer event sequence we have described in the paragraphs above. The contents of Accumulator A have been inverted in order to complement a single bit. **It would be just as reasonable to complement the contents of Accumulator B.** This could be accomplished by the following single instruction:

> COM B   COMPLEMENT ACCUMULATOR B

The above instruction, when converted into object code and executed, inverts all eight bits of Accumulator B.

Even though the contents of Accumulator A or Accumulator B could be complemented with equal ease, for the rest of this chapter we are going to confine ourselves to complementing the contents of Accumulator A, since the choice of which Accumulator is to be complemented is not relevant to the discussion at hand.

IMPLEMENTING THE TRANSFER FUNCTION

The COM A instruction inverts every bit of Accumulator A.

BIT DATA

The COM A instruction does not specify which bit of Accumulator A represents the signal to be inverted. This specification is implied by the way in which data is input to, and output from, the microcomputer system.

DETERMINING DATA SOURCES AND DESTINATIONS

How will Accumulator data be input to, and output from, the microcomputer system? In answering this question, we touch on one of the fundamental strengths (and complexities) of microcomputers — their flexibility.

The input signal and the inverted output signal are just what their names imply — they are signals. But to the microcomputer system, they are "external logic". Information transfers between external logic and the microcomputer system are referred to generically as Input/Output (or I/O).

EXTERNAL LOGIC AS THE SOURCE OR DESTINATION

During any programmed I/O operation, recall that the microcomputer is master and external logic is slave. This means that the microcomputer must indicate the direction of the I/O operation (input or output), and must identify the external logic being accessed.

INPUT/OUTPUT

External logic will decode a specific memory address as an enable strobe, so that I/O is handled as though it were a memory read or write. **Suppose the label INVD is being used in the assembly language source program to identify the signal being inverted. This is the instruction sequence which will reproduce the signal inverter:**

I/O IN MEMORY ADDRESS SPACE

LDA A	INVD	LOAD ACCUMULATOR A FROM INVD
COM A		COMPLEMENT ACCUMULATOR A
STA A	INVD	STORE ACCUMULATOR A CONTENTS AT INVD

In terms of microcomputer devices, this is the microcomputer configuration implied:
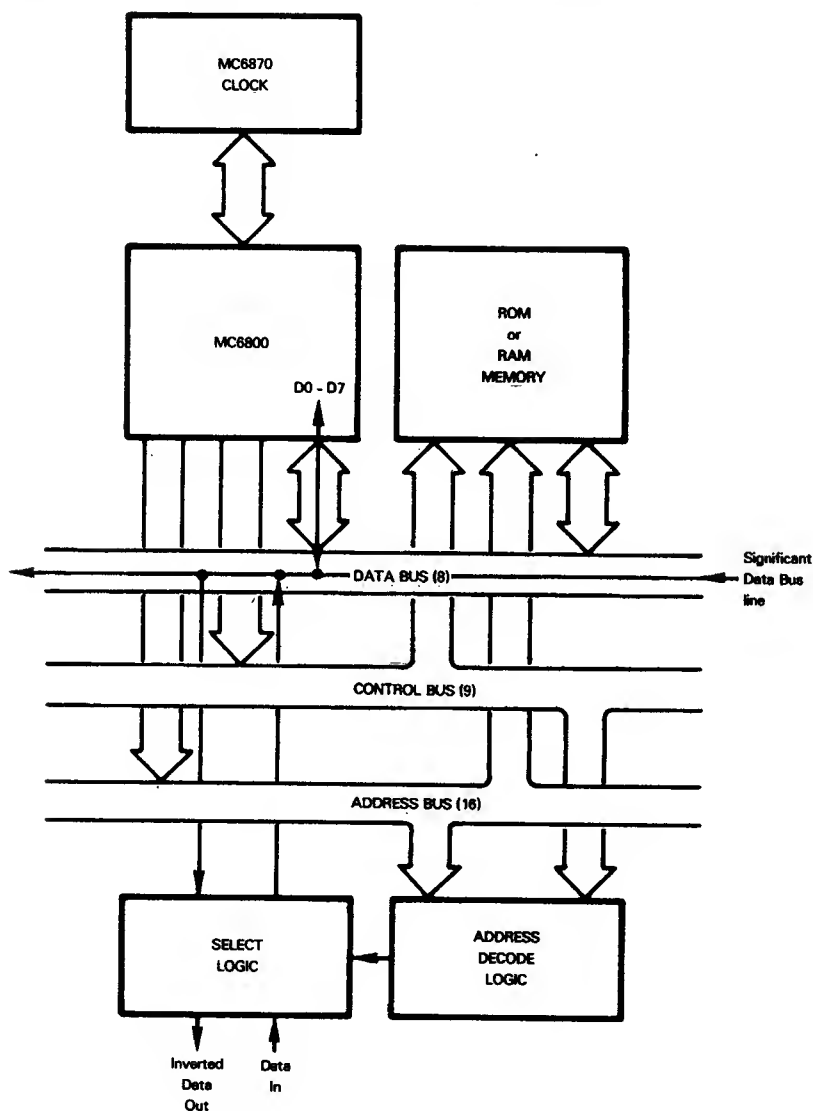
When the LDA A instruction is executed, "Address Decode Logic" causes "Select Logic" to transmit the "Data In" signal to the Data Bus.

There are eight Data Bus lines; the number of the line to which the "Data In" signal is connected becomes the significant bit number within Accumulator A. When the LDA A instruction has completed execution, the contents of the Data Bus will be in Accumulator A.

Next the COM A instruction is executed. This instruction causes every bit of Accumulator A to be complemented.

When the STA A instruction is executed, the contents of Accumulator A are output to the Data Bus. "Address Decode Logic" then causes "Select Logic" to output the contents of a single Data Bus line — which becomes the inverted "Data Out" signal.

Because the "Select Logic" has "Data In" and "Data Out" signals connected to the same line of the Data Bus, "Data Out" is the complement of "Data In"; and the signal inverter has been simulated.

ROM or RAM memory must be present in the microcomputer system, because the object codes for the three instructions must be stored in, and fetched out of memory.

Consider the object code in detail. The three source program instructions become object code as follows:

OBJECT CODE
INTERPRETATION

PROGRAM
MEMORY

Control signals are output on Control Bus. Address decode logic receives this signal and triggers Data in to select logic.

LDA A	B6
INVD	XX
	YY
COM A	43
STA A	B7
INVD	XX
	YY

Complement Accumulator A

Control signals are output on Control Bus. Address decode logic receives this signal and triggers inverted Data out from select logic.

16-bit address, represented by XXYY, output on Address Bus. Address Decode logic decodes just one 16-bit combination as a "select true".

The program memory addresses of the bytes within which the object codes are stored are not important. However, no memory byte; ROM or RAM, can have the address represented by XXYY, since external logic is selected by this address.

In most MC6800 microcomputer configurations address decode logic and select logic will be provided by an MC6820 Peripheral Interface Adapter (PIA). The MC6820 PIA is the MC6800 microcomputer system's standard parallel interface logic device. When incorporated into the signal inverter, a single PIA I/O port pin will receive the "Data In" signal and a single I/O port pin will output the inverted signal. Suppose an MC6820 PIA I/O port given the label INVDI has pin 4 assigned to receive the "Data In" signal, and another MC6820 PIA I/O port, labeled INVDO, has pin 4 assigned to output the inverted signal:

7 6 5 4 3 2 1 0 ◄── Bit No.

I/O Port INVDI

Data In to be inverted

7 6 5 4 3 2 1 0 ◄── Bit No.

I/O Port INVDO

Inverted Data Out
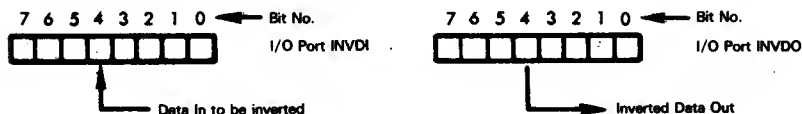
We can use a technique known as "masking" in order to invert a single I/O port pin, leaving all other pins alone. In this instance, masking may be illustrated as follows:
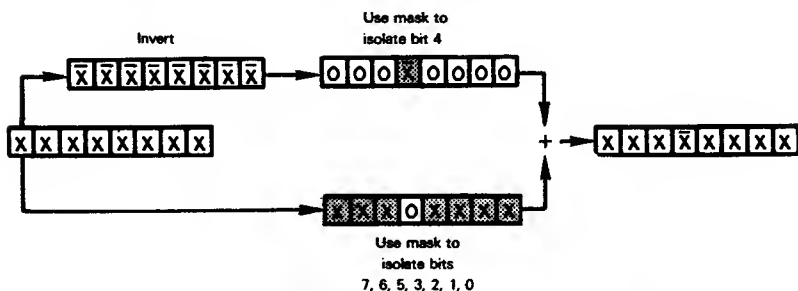
In the illustration above, X represents any binary digit; $\overline{X}$ represents its complement.

The following instruction sequence will invert pin 4, leaving all other pins as they were:

LDA A	INVDI	INPUT TO ACCUMULATOR A FROM I/O PORT INVD
COM A		COMPLEMENT ACCUMULATOR A
AND A	#$10	ISOLATE BIT 4
LDA B	INVDO	INPUT TO ACCUMULATOR B FROM I/O PORT INVD
AND B	#$EF	CLEAR BIT 4
ABA		ADD ACCUMULATOR B TO A
STA A	INVDO	OUTPUT ACCUMULATOR A TO I/O PORT INVD

A # sign beginning an operand field identifies immediate data; consequently the immediate form of the instruction is being used. A $ sign preceding a number identifies the number as being hexadecimal.

Thus, $EF represents the binary value:

$$\underbrace{1\ 1\ 1\ 0}_{E}\ \underbrace{1\ 1\ 1\ 1}_{F}$$

In terms of registers' contents, this is what happens when the above instruction sequence is executed (again X represents any binary digit):

		I/O PORT	ACCUMULATOR A	ACCUMULATOR B
		X X X X X X X X	?	?
LDA A	INVDI	X X X X X X X X	X X X X X X X X	?
COM A		X X X X X X X X	$\overline{X}\ \overline{X}\ \overline{X}\ \overline{X}\ \overline{X}\ \overline{X}\ \overline{X}\ \overline{X}$	?
			$\wedge$ 0 0 0 1 0 0 0 0	
AND A	#$10	X X X X X X X X	0 0 0 $\overline{X}$ 0 0 0 0	?
LDA B	INVDO	X X X X X X X X	0 0 0 $\overline{X}$ 0 0 0 0	X X X X X X X X
				$\wedge$ 1 1 1 0 1 1 1 1
AND B	#$EF	X X X X X X X X	0 0 0 $\overline{X}$ 0 0 0 0	X X X 0 X X X X
			$+$ X X X 0 X X X X	
ABA		X X X X X X X X	X X X $\overline{X}$ X X X X	X X X 0 X X X X
STA A	INVDO	X X X $\overline{X}$ X X X X	X X X $\overline{X}$ X X X X	X X X 0 X X X X

EVENT TIMING

Within any digital logic implementation, events may be timed synchronously, based on a clock signal:
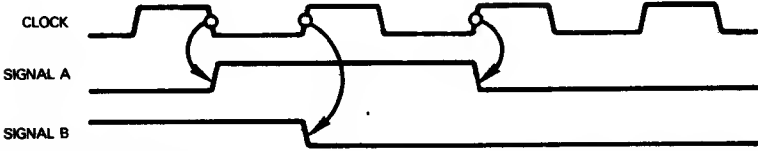
or asynchronously, based upon an output signal from one device changing state and thus triggering another device's state change:
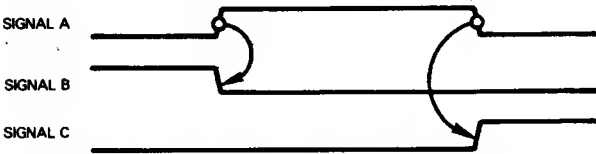
Simple gates, however, are continuous devices. Consider the following simple logic sequence:
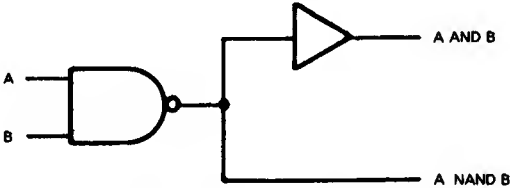
The signal inverter continuously inverts its input; a gate settling time of perhaps 10 nanoseconds is the only lag between input and output signal state changes.

Within a microcomputer system, however, three instructions must be executed before an output signal can reflect an input signal's state change.

In the unlikely event that the microcomputer system is emulating an inverter and doing nothing else, the inverter instruction sequence could be continuously re-executed as follows:

```
LOOP    LDA  A    INVD    LOAD ACCUMULATOR A FROM INVD
        COM A             COMPLEMENT ACCUMULATOR A
        STA  A    INVD    STORE ACCUMULATOR A CONTENTS AT INVD
        JMP       LOOP    RE-EXECUTE THE SIGNAL INVERTER SEQUENCE
```

Depending on the microcomputer clock frequency, it will take approximately 12 microseconds to execute the signal inverter instruction loop once; providing the period between input signal state changes is never less than 12 microseconds, the microcomputer implemented signal inverter will always work. But **there may be a delay of up to 12 microseconds between an input signal changing state and the output signal following suit.** This may be illustrated as follows:

① = LDA instruction execution
② = COM instruction execution
③ = STA instruction execution
④ = JMP instruction execution
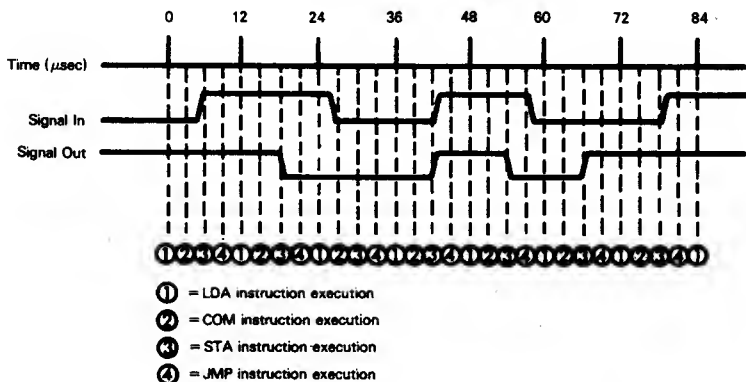
In the above illustration, the four instructions have been shown dividing 12 microseconds equally, so that each instruction is executed in 3 microseconds. In reality, this is not the case. Chapter 6 gives instruction execution times; you will see that the COM instruction, for example, requires considerably less time to execute than any of the other three instructions. We will overlook this detail for the moment in order to concentrate on the concept at hand — which is that **we must pay careful attention to event sequences within the microcomputer system.**

Irrespective of when and how "Signal In" changes state, it is the state of "Signal In" at time ① (when the LDA instruction is executed) which is transported, as a binary digit, into the microcomputer system.

The actual binary digit inversion occurs at time ② .

The inverted binary digit is converted into "Signal Out" at time ③ , when the STA instruction is executed.

Thus, "Signal Out" timing may differ considerably from "Signal In" timing.

More serious problems arise when the signal inverter instruction sequence is just one small part of a larger microcomputer program. Under these circumstances, many milliseconds may elapse between repeated executions of the inverter instruction sequence. If you leave it to chance, signal inversions may be completely missed. At very best there may be considerable delays between the input signal changing state and the output signal following suit. This situation is illustrated as follows:
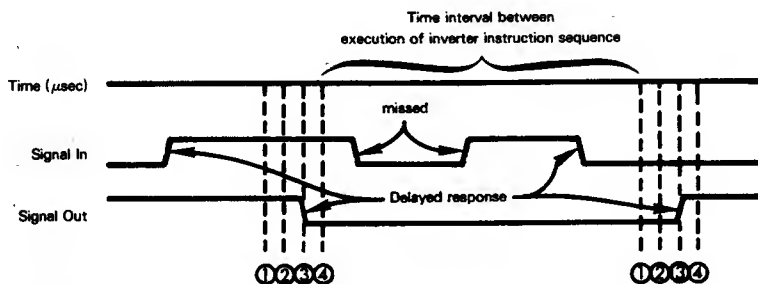
Again ①, ②, ③ and ④ identify LDA, COM, STA and JMP instructions' execution, respectively.

Having stressed the importance of timing in a microcomputer system, plus the consequences of poor timing, we will drop the subject for the moment. This is because **timing problems largely evaporate when you simulate entire logic sequences as opposed to individual devices.** Therefore solutions to timing problems should be looked at in the context of an entire logic simulation; and we have not yet progressed that far.

BUFFERS, AMPLIFIERS AND SIGNAL LOADS

Having looked at timing, we will now turn to some other fundamental digital logic concepts.

A signal buffer increases the signal current level: | BUFFER |

Buffer

An amplifier driver increases the signal voltage level: | AMPLIFIER |

Amplifier, driver

Every device has a well defined fan out. Fan out defines the num- **FAN OUT**
ber of parallel loads that may be connected to an output signal:
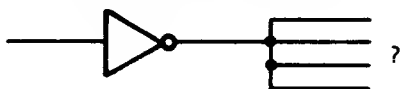
Logic devices will also have specified fan in, which indicates the number **FAN IN**
of parallel loads which may be connected to a device input:

What happens to these concepts once your logic disappears into a microcomputer program? The answer is simple: these concepts disappear — along with digital logic.

Now at the actual pins of a physical microcomputer device, fan in and fan out remain legitimate concepts; signals travelling between pins of individual microcomputer devices may need to be amplified and buffered. For example, an MC6800 CPU device's fan out may be as little as one or two Transistor-Transistor Logic (TTL) loads; that means if more than one or two similar devices connect to an output signal, the output signal will have insufficient power to transmit usable signals to all connected devices. Therefore for all but the simplest microcomputer configurations, bus lines will have to be buffered.

FAN IN
FAN OUT
TTL LOADS
SIGNAL BUFFERING

When determining whether your bus lines need to be buffered, do not ignore leakage current. For example, if you have sixteen ROM devices connected to the System Bus, and only one device can be selected (and therefore connected) at any time, do not assume that the total signal load is due to the selected ROM. The fifteen unselected ROM devices will each tap off some leakage current; that alone may require System Bus buffering.

LEAKAGE CURRENT

Within a microcomputer program, however, when logic is totally represented by a microcomputer instruction sequence, you are dealing exclusively with binary digits — never with voltage or current levels. Fan in is infinite, since the status of a binary digit may be the result of any number of logical computations. Fan out is infinite, since you can read the status of a binary digit as often as you want. Buffers and amplifiers are meaningless, since a binary digit has no qualities equivalent to voltage or current. A binary digit offers pure, finite resolution.

Take another look at the signal inverter, as simulated by a microcomputer.

We will take a giant conceptual step and assume that the signal inverter is buried within a logic sequence, such that no input or output signal is generated at any microcomputer device pin. In other words, the signal inverter becomes a small part of a larger transfer function.

The input to the signal inverter is a binary digit created by some previous logic.

The output from the signal inverter is another binary digit which becomes input to subsequent logic.

Logic external to the microcomputer system does not supply the inverter input as a signal arriving at a microcomputer device pin, nor does the inverted signal get transmitted to external logic via a microcomputer device pin. Rather, the interface between external

logic and the microcomputer system occurs at some point significantly before and beyond the signal inverter. **Our signal inverter may now be represented by these same three instructions:**

```
LDA  A   INVD        LOAD ACCUMULATOR A FROM INVD
COM A                COMPLEMENT ACCUMULATOR A
STA  A   INVD        STORE ACCUMULATOR A CONTENTS AT INVD
```

The source and destination become data memory bits; this may be illustrated as follows:
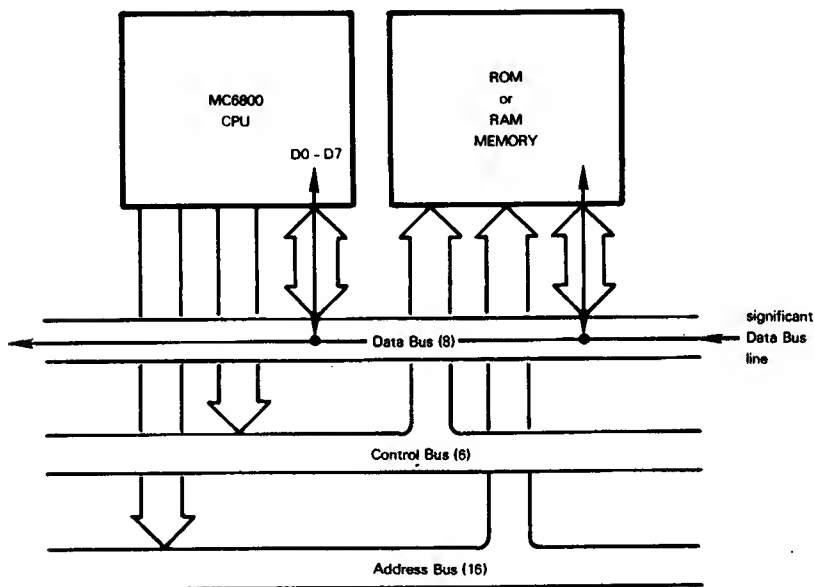
In terms of memory and CPU register contents, the signal inverter sequence proceeds as follows:

①

②

③
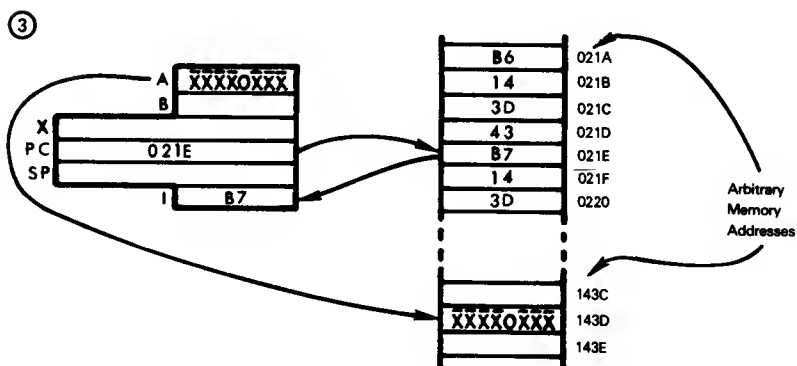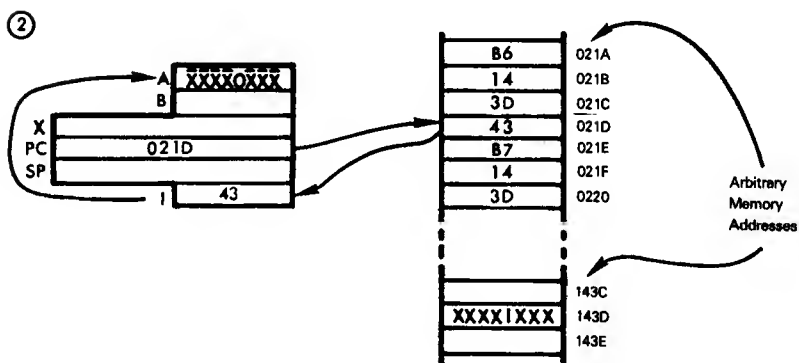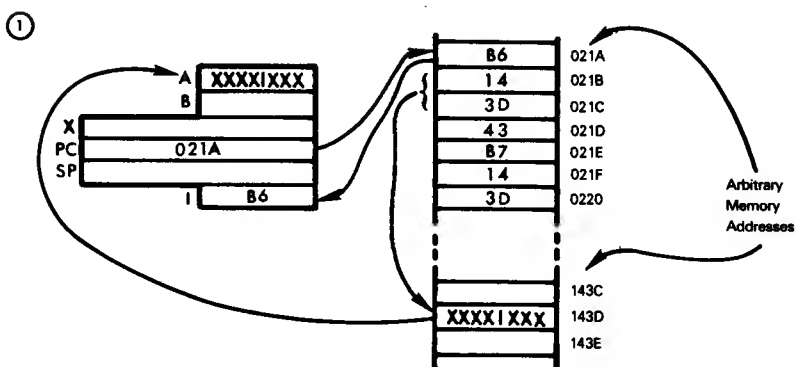
With regard to the above illustration, the letters A and B identify the two CPU Accumulators. PC represents the Program Counter. SP represents the Stack Pointer. I represents the Instruction register.

The contents of data memory byte $143D_{16}$ and Accumulator A are represented in binary format. X represents any binary digit. Note that we have arbitrarily selected bit 3 to be the significant bit.

In step ①, the LDA instruction is executed. This instruction causes the contents of data memory byte $143D_{16}$ to be loaded into Accumulator A.

During step ②, the COM instruction is executed. This causes the contents of Accumulator A to be complemented.

During step ③, the contents of Accumulator A are loaded back into memory byte $143D_{16}$.

Signal inversion has been simulated by inverting the contents of bit 3 (along with every other bit) of data memory byte $143D_{16}$.

Where does the inverter's input come from? A data memory bit. **Let us suppose, to illustrate a point, that the inverter input is the OR of eight signals.** We could not wire-OR these eight signals to create an inverter input as follows:

<table>
<tr><td>FAN IN IN
MICROCOMPUTER
PROGRAMS</td></tr>
</table>
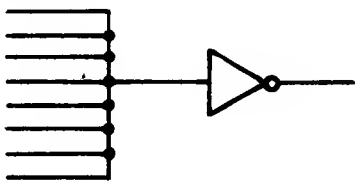
because that would exceed the fan in capacity of the signal inverter.

But **presuming the eight signals are represented by the eight binary digit contents of the Accumulator, we would have no trouble generating the inverter input via the following logic sequence:**
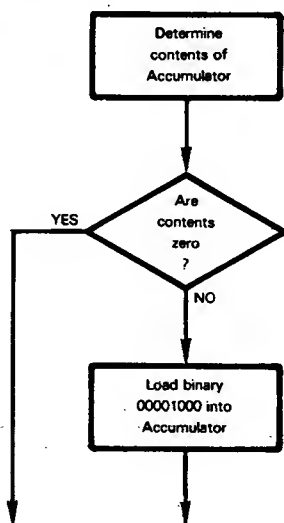
The fan in logic is implemented by this instruction sequence:

ASSUME THAT EIGHT SIGNALS ARE TO BE REPRESENTED BY THE EIGHT BITS
OF ACCUMULATOR A

```
         LDA  A    INVD    LOAD EIGHT SIGNALS INTO ACCUMULATOR A
         BEQ       NEXT    ACCUMULATOR A HOLDS 0. SIGNAL IN MUST BE 0
         LDA  A    #8      ACCUMULATOR A HOLDS NONZERO. SIGNAL IN MUST BE 1
NEXT     STA  A    INVD    CREATE APPROPRIATE OUTPUT
```

Note that the second LDA instruction is identified as an immediate load since there is a # sign in the operand field. We need not bother specifying the 8 as hexadecimal since decimal 8, the default option, has the same value as hexadecimal 8.

The above instruction sequence is a direct microcomputer program implementation of the eight signal wire-OR. Let us examine how the instruction logic works.

We are going to assume that the eight input signals are initially represented by the status of the eight Accumulator binary digits:
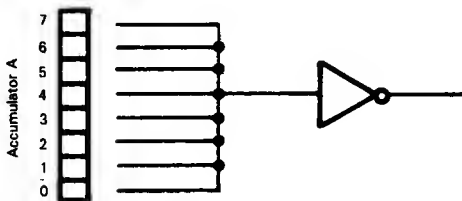
We are further going to assume that, in keeping with the prior illustration, bit 3 of the data byte will ultimately be the significant inverter signal bit.

Since the inverter input is the wire-OR of eight signals, program logic must set bit 3 of Accumulator A to 1 if any Accumulator bit is nonzero; bit 3 of Accumulator A must be set to 0 if all Accumulator bits are zero. The contents of Accumulator A are then stored in the data memory byte represented by label INVD. With regard to the previous illustration, INVD would be a label representing memory byte $143D_{16}$.

This is how the four-instruction sequence illustrated above works:

We assume that the initial eight signals have their status at some memory location represented by the label INVD. The first LDA instruction loads these eight signal statuses into Accumulator A. The Zero and Sign bits of the Status register are set or reset during the course of the LDA instruction's execution to represent the contents of Accumulator A and the contents of the high order bit of Accumulator A, respectively.

After execution of the LDA instruction, if the Zero status is 1, then bit 3 of Accumulator A must already be 0, which is what we want it to be. No operation is required and we jump to the STA instruction.

If the Zero bit was 0, then one or more bits of Accumulator A are nonzero. The LDA A #8 instruction loads a 1 into bit 3 of the Accumulator:
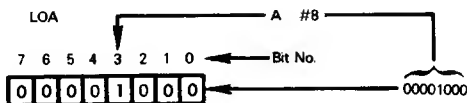
Finally the STA instruction is executed to load the inverter input signal into the appropriate data memory byte.

Now suppose the inverter output is distributed to numerous subsequent devices. The following logic represents fan out that is not feasible:
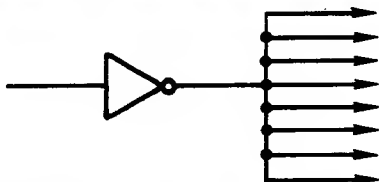
Within a microcomputer program, the whole concept of fan out disappears. The inverter output may be accessed an indefinite number of times by the simple re-execution of an LDA instruction:

FAN OUT IN MICROCOMPUTER PROGRAMS

```
LDA A    INVD    LOAD INVERTER OUTPUT INTO ACCUMULATOR A
 -
 -
 -
LDA A    INVD    LOAD INVERTER OUTPUT INTO ACCUMULATOR A
 -
 -
 -
LDA A    INVD    LOAD INVERTER OUTPUT INTO ACCUMULATOR A
 -
 -
 -
LDA A    INVD    LOAD INVERTER OUTPUT INTO ACCUMULATOR A
 -
 -
 -
LDA A    INVD    LOAD INVERTER OUTPUT INTO ACCUMULATOR A
```

What about amplifiers and buffers? Clearly within the context of binary data stored in memory, they have no meaning. If amplifiers and buffers are present because of the electrical characteristics of the memory and processor chips, that has nothing to do with the logic function being implemented by a microcomputer program.

MICROCOMPUTER SIMULATION OF 7404/05/06 HEX INVERTERS

These three hex inverters differ only in their electrical characteristics:

The 7404 is a simple hex inverter.

The 7405 is a hex inverter with open collector outputs.

The 7406 is a hex inverter buffer/driver with open collector, high voltage outputs.

Since these three devices differ only in their electrical characteristics, within a microcomputer assembly language simulation they are identical. Let us look at the 7404. It consists of six independent signal inverters, which may be illustrated as follows:

$$Y = \overline{A}$$
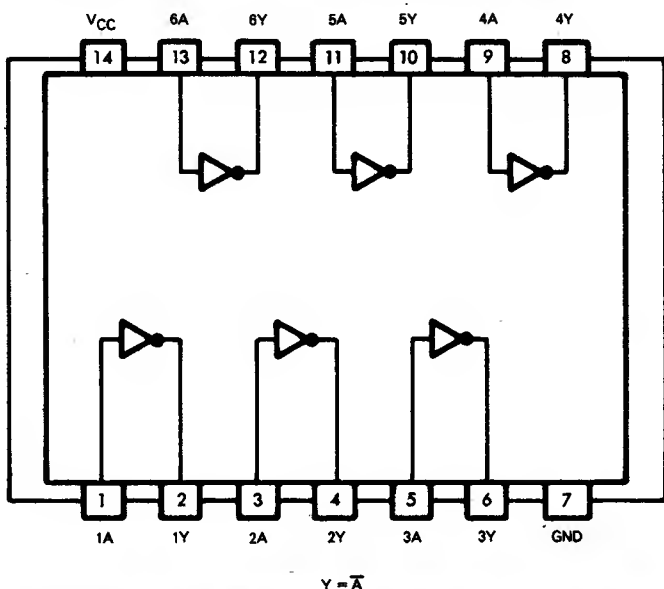
The instruction sequence to represent a hex inverter is identical to the three-instruction, single signal inverter instruction sequence, because MC6800 microcomputers are eight-bit parallel devices. Whether you like it or not, this inverter instruction sequence inverts eight independent binary digits. Hex inverters may therefore be represented within a microcomputer instruction sequence as follows:

```
LDA  A    INVD        LOAD ACCUMULATOR A FROM INVD
COM A                 COMPLEMENT ACCUMULATOR A
STA  A    INVD        STORE ACCUMULATOR A CONTENTS TO INVD
```

We will arbitrarily identify significant bits, as implied by the hex inverter, as follows:
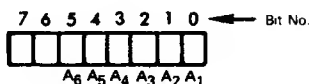
Note that the above selection of significant bits is completely arbitrary. There is absolutely no practical or philosophical argument favoring any one bit assignment as compared to any other.

MICROCOMPUTER SIMULATION OF
7408/09 QUADRUPLE TWO-INPUT POSITIVE
AND GATES

These two devices provide four independent, two-input, one output AND gates, which may be illustrated as follows:

$$Y = A \cdot B$$
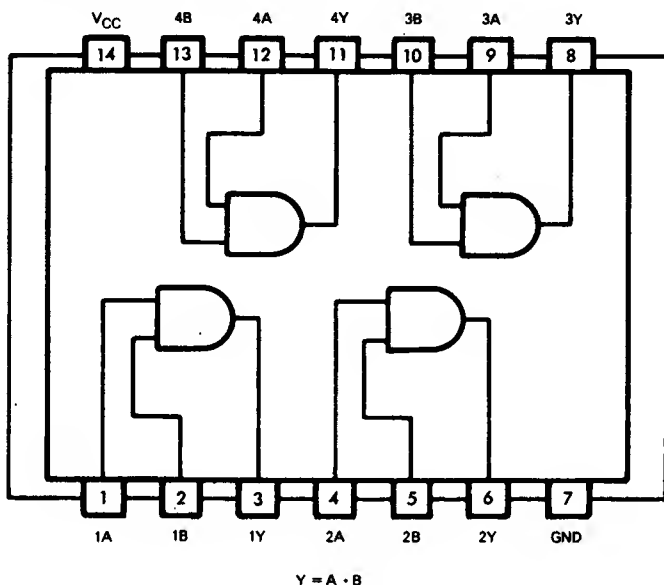
The 7409 has open collector outputs, which differentiates it from the 7408. This difference has no meaning in a microcomputer program simulation, therefore the two devices can be looked on as being identical.

TWO INPUT FUNCTIONS

From the microcomputer programmer's point of view, the most significant difference between a 7408 AND gate and a 7404 inverter is not the logic function; rather it is the fact that a 7408 is a two-input device. Conceptually, we might imagine a 7404 being simulated in one of the two following ways:

1) The eight input signals are loaded into the CPU Accumulator register. Each even-numbered bit is ANDed with the bit to its right. The result is deposited in the even-numbered bit for each bit pair:
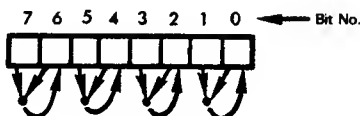
2) The two sets of four inputs are loaded into the CPU Accumulator and one other register. The result is returned in the Accumulator:
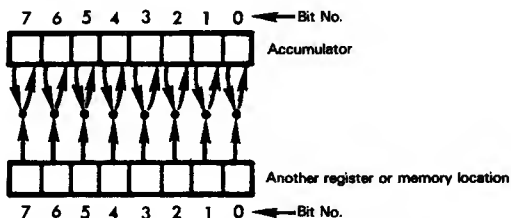
Upon examining the MC6800 microcomputer instruction set, you will find that the second method of simulating a 7408 is the natural one. This is the required instruction sequence:

```
LDA A    SRCA    LOAD FIRST SET OF INPUTS, FROM SRCA
AND A    SRCB    AND WITH SRCB. THE RESULT IS IN A
STA A    DST     SAVE RESULT IN DST
```

If the use of labels SRCA, SRCB and DST still confuses you, let us take a minute to clarify them. Eventually you will have some amount of memory which may vary from as little as 256 bytes to as much as 65,536 bytes. Each of the labels SRCA, SRCB and DST identifies one memory byte. At the time you are writing the

**SOURCE
PROGRAM
LABEL
ASSIGNMENTS**

source program, the exact memory byte identified by each label is unimportant. When you eventually assemble your source program, the assembler listing will print a memory map. The memory map will identify the exact memory byte associated with each label you have used. By examining the memory map, you will be able to determine whether or not all label assignments are valid. If any label assignments are invalid, you will have to take appropriate action. Appropriate action may involve adding more memory to your microcomputer configuration, or you may have to rewrite your source program, so that it makes more effective use of the memory you have.

The problem of labels and memory allocations is irrelevant at the present level of discussion. Simply imagine every label as addressing one specific memory byte. Do not worry about which memory byte will eventually be addressed, and your problem will disappear.

The 7408 simulation instruction sequence illustrated above by no means represents the only way in which a 7408 may be simulated.

Accumulator B could replace Accumulator A:

```
LDA B    SRCA    LOAD FIRST SET OF INPUTS FROM SRCA
AND B    SRCB    AND WITH SRCB. THE RESULT IS IN B
STA B    DST     SAVE RESULT IN DST
```

THE MICROCOMPUTER SIMULATION OF A
7411 TRIPLE, THREE-INPUT, POSITIVE AND GATE

The principal difference between the 7411 AND gate and the 7408 AND gate is
the number of input signals. The 7411 generates three output signals, each of
which is the AND for three inputs:

$$Y = A \cdot B \cdot C$$
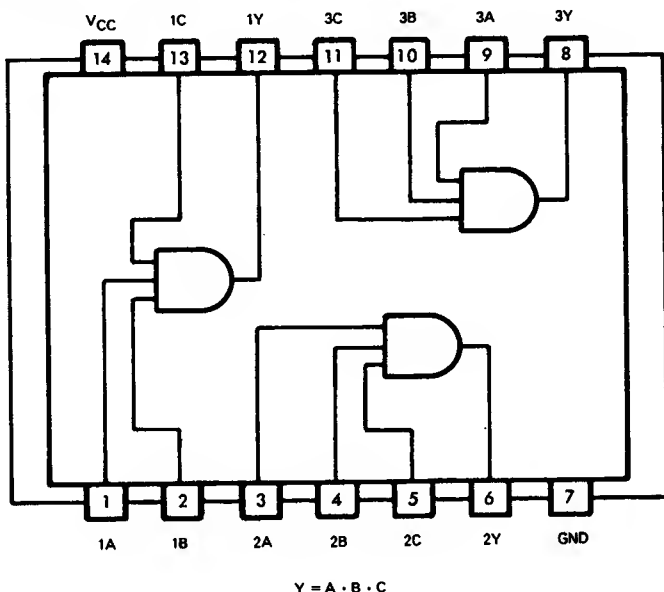
THREE INPUT FUNCTIONS

The fact that the MC6800 instruction set has many memory reference instruc-
tions makes multiple input functions easy to handle. Here is the three input AND in-
struction sequence:

ONE	LDA A	SRCA	LOAD FIRST SET OF INPUTS FROM SRCA
TWO	AND A	SRCB	AND WITH SECOND SET OF INPUTS IN SRCB. THE RESULT IS IN A
THRE	AND A	SRCC	AND RESULT IN A WITH SRCC. THE FINAL RESULT IS IN A
FOUR	STA A	DST	SAVE THE RESULT IN DST

When instruction ONE executes, an 8-bit value is loaded into Accumulator A from the memory
byte addressed by label SRCA. We will assume that AND gate inputs are represented as follows:
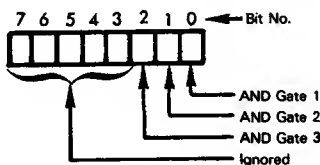
Understand that the assignment of data bits illustrated above is completely arbitrary. It is only necessary that all subsequent inputs be consistent.

Instruction TWO ANDs the contents of the memory byte addressed by SRCB with the contents of Accumulator A, leaving the result in Accumulator A, as follows:
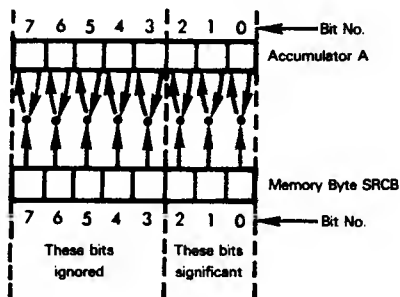
Instruction THRE performs the second AND operation. This time the AND occurs between Accumulator A and the memory byte addressed by SRCC. The Accumulator initially holds the result of the AND with SRCB, as illustrated above. After instruction THRE has executed, the AND of three inputs is in Accumulator A.

Instruction FOUR returns the final result to a memory byte addressed by the label DST. The 7411 AND gate simulation is complete.

THE MICROCOMPUTER SIMULATION OF A 7474 DUAL, D-TYPE, POSITIVE EDGE TRIGGERED FLIP-FLOP WITH PRESET AND CLEAR

Before looking at the 7474 flip-flop in particular, let us consider flip-flops in general. First a few definitions.

A DIGITAL LOGIC DESCRIPTION OF FLIP-FLOPS

A flip-flop is a bistable logic device, that is, a device which may exist in one of two stable conditions. 7474 type flip-flops have two outputs, Q and $\overline{Q}$; thus the two bistable conditions may be represented as follows:
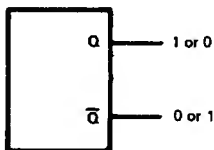
A clock signal causes the flip-flop to change from one bistable condition to the other. A positive edge triggered flip-flop changes upon sensing a zero-to-one transition of the clock signal:
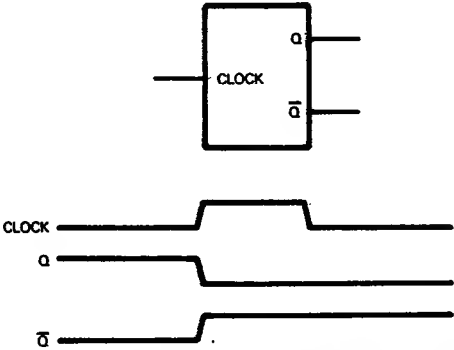
A negative edge triggered flip-flop changes state upon sensing a one-to-zero clock signal transition:

NEGATIVE EDGE TRIGGER
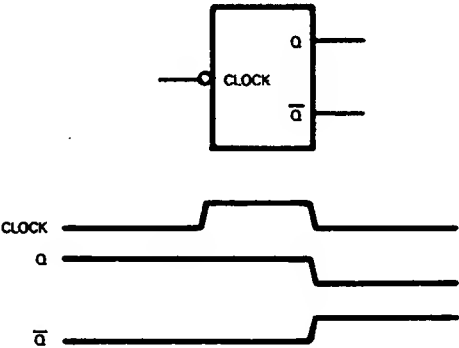
A JK flip-flop preconditions the Q and $\overline{Q}$ outputs which will be generated by the next clock edge trigger as follows:

JK FLIP-FLOP

STATUS OF J AND K AT CLOCK SIGNAL		OUTPUTS GENERATED AT CLOCK SIGNAL	
J	K	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
0	0	Stay as you were.	
1	1	Change state regardless of previous state.	

2-25
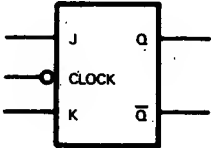
In the above table, "clock signal" will be a zero-to-one transition for a positive edge-triggered device; it will be a one-to-zero transition for a negative edge triggered device. This definition of "clock signal" also applies to the D type flip-flop described next.

CLOCK SIGNAL

By inverting a J input in order to generate the K input, a D type flip-flop is created. These are the D type flip-flop characteristics that result:

D TYPE FLIP-FLOP

STATUS OF J AND K AT CLOCK SIGNAL		OUTPUTS GENERATED AT CLOCK SIGNAL	
$J=D$	$K=\overline{J}$	Q	$\overline{Q}$
1	0	1	0
0	1	0	1
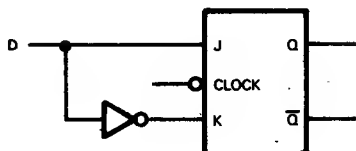
Here is a positive edge triggered, D type flip-flop timing diagram:
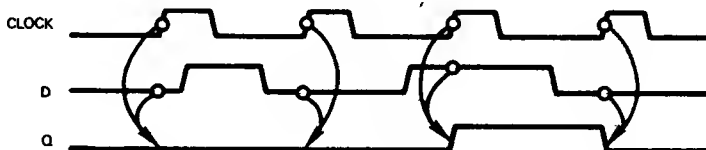
A D type flip-flop therefore will always output the input conditions that existed at the previous clock pulse.

The presence of a Preset input means that the flip-flop may be forced to output $Q = 1$ and $\overline{Q} = 0$. Preset true forces this condition.

FLIP-FLOP PRESET

A Clear input is the opposite of a Preset input. When true, the Clear input forces $Q = 0$ and $\overline{Q} = 1$.

FLIP-FLOP CLEAR

Combining the definitions given above, this is what we get for a 7474 type flip-flop:

FUNCTION TABLE

INPUTS				OUTPUTS	
1PR or 2PR	1CLR or 2CLR	1CK or 2CK	1D or 2D	1Q or 2Q	1$\overline{Q}$ or 2$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$
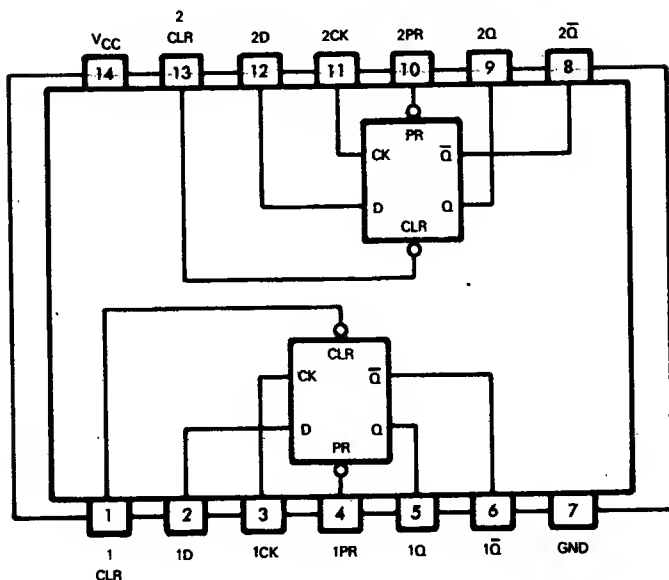
In the function table above, ↑ represents a clock zero-to-one transition. H* signifies an unstable state. Q_0 is the previous state for Q. X signifies "Don't care".

AN ASSEMBLY LANGUAGE SIMULATION OF FLIP-FLOPS

Now our first problem, when trying to simulate a 7474 flip-flop, is the fact that there is no clock signal within a microcomputer instruction set. Instead we must assume that events are triggered by execution of an appropriate instruction, rather than a clock signal transition.

How will we represent outputs Q and Q̄? Two bits of memory could be used to represent these two outputs:

```
7 6 5 4 3 2 1 0 ◄── Bit No.
┌─┬─┬─┬─┬─┬─┬─┬─┐
│ │ │ │ │ │ │ │ │
└─┴─┴─┴─┴─┴─┴─┴─┘
                ├──── Represent Q
                └──── Represent Q̄
```
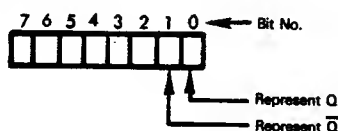
Since we are dealing with data, not signals, Q̄ is redundant. The single flip-flop therefore devolves to one memory bit. A 7474 device, since it contains two flip-flops, devolves to two memory bits, one for each flip-flop implemented on the chip.

There is nothing surprising about this conclusion. Each bit of a microcomputer's read/write memory is a simple, bistable element; it could, indeed, be a flip-flop.

The logic of a 7474 flip-flop may be represented by instructions that clear s memory bit, set the memory bit to 1, or store sn unknown binary digit in the memory bit.

Suppose memory bits are assigned as follows:

```
7 6 5 4 3 2 1 0 ◄── Bit No.
┌─┬─┬─┬─┬─┬─┬─┬─┐
│ │ │ │ │ │ │ │ │
└─┴─┴─┴─┴─┴─┴─┴─┘
              ├──── First flip-flop
              └──── Second flip-flop
                    Unused
```
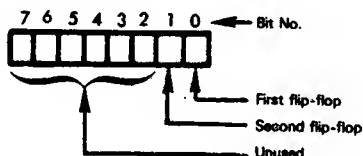
The 7474 function table now becomes these instructions:

Preset	Clear	D	First flip-flop	Second flip-flop
L	H	X	LDA A FLP	LDA A FLP
H	H	H	ORA. A #1	ORA A #2
			STA A FLP	STA A FLP
H	L	X	LDA A . FLP	LDA A FLP
H	H	L	AND A #2	AND A #1
			STA A FLP	STA A FLP
L	L	X	Does not apply	

With regard to the table above, the LDA instruction acts on Accumulator A contents as follows:

```
        LDA  A   #DATA          Accumulator A
             Any 8-bit         ┌──────────────┐
             data value        └──────────────┘
```
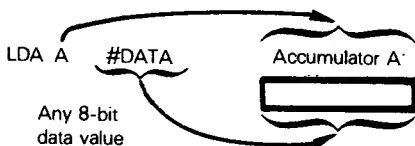
The STA A instruction stores the resulting Accumulator A contents in a memory word identified by the label FLP. Bits 0 and 1 of the memory word identified by FLP are presumed equivalent to the 2 flip-flops of the 7474 device.

MICROCOMPUTER SIMULATION OF FLIP-FLOPS IN GENERAL

In conclusion, a flip-flop becomes a single bit of read/write memory within a microcomputer system.

Within a microcomputer syetem, all flip-flops are the same. Flip-flop logic reduces to these four questions:

1) When do I execute an instruction to set a memory bit to 1?

2) When do I execute an instruction to reset a memory bit to 0?

3) When do I execute an instruction to store a binary digit in a memory bit?

4) When do I execute an instruction to read the contents of a memory bit?

THE MICROCOMPUTER SIMULATION OF REAL TIME DEVICES

There are two types of real time devices that we will look et: the one-shot (including monostable multivibrators) end the master-slave flip-flop. Specifically, these devices will be described:

● The Signetics 555 monostable multivibrator

● The 74121 monostable multivibrator

● The 74107 dual J-K master-slave flip-flop with Clear

A one-shot is a device which generates a signal pulse with a specific time period: ONE-SHOT

Period of signal pulse
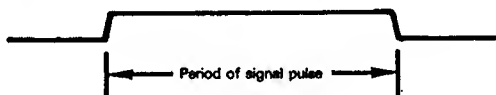
A monostable multivibrator is a device with one stable, or passive state. It produces one-shot output signals, as illustrated above, where the pulse is in the unstable, or active state: MONOSTABLE MULTIVIBRATOR

Active State
Passive State
Period of signal pulse
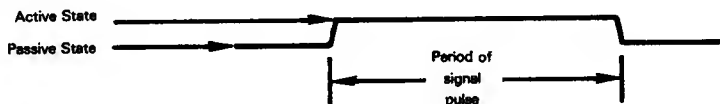
The device is a "multivibrator" because it can output a continuous stream of signals — much like a clock signal. In other words, a multivibrator output consists of a continuous stream of one-shot signals.

The time period of the signal pulse is a real time value — it is a finite number of microseconds, or milliseconds, or even seconds.

A master-slave flip-flop is a flip-flop which generates output signels based on the condition of input signals at some earlier time. Again we encounter a real time value — the delay between inputs and outputs. MASTER-SLAVE FLIP-FLOP

THE 555 MONOSTABLE MULTIVIBRATOR

The Signetics 555 monostable multivibrator may be illustrated as follows:
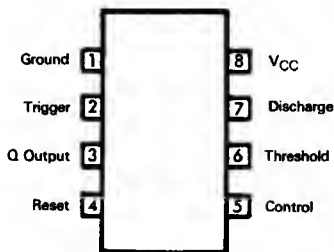
The negative edge of a clock signal at the Trigger input (pin 2) causes a negative-to-positive transition at the Output Q. The duration of the high level output at Q is controlled by a resistor/capacitor circuit connected to the Discharge and Threshold pins (7 and 6, respectively).

Reset is a standard reset input; a low input will hold the Q output low.

The Control pin is used to control voltage within the multivibrator; it is not significant to an overall understanding of how the 555 device works.

The ground and power pins (1 and 8, respectively) are self-explanatory.

Here is one way in which the 555 monostable multivibrator may be configured:
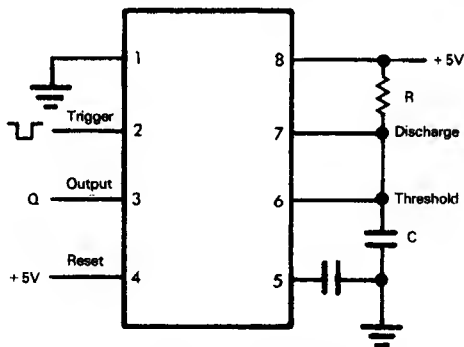
As soon as a high-to-low signal level is sensed at the Trigger input, the capacitor between pin 6 and ground charges. Signal levels at the threshold and discharge pins, as controlled by the resistor R and the capacitor C, control the period for which Q will output high. This time period is given by the following equation:

$$T = 1.1RC$$

Where T is time in seconds
R is resistance in Megohms
C is capacitance in microfarads

An output signal pulse is generated as follows:
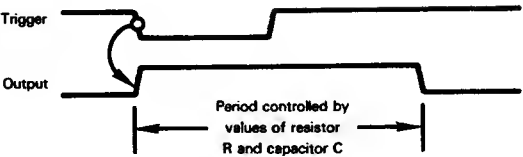
THE 74121 MONOSTABLE MULTIVIBRATOR

The 74121 monostable multivibrator may be illustrated as follows:

FUNCTION TABLE

INPUTS			OUTPUTS		
A1	A2	B	Q	$\overline{Q}$	
L	X	H	L	H	
X	L	H	L	H	Monostable
X	X	L	L	H	outputs
H	H	X	L	H	
H	↓	H	⊓	⊔	
↓	H	H	⊓	⊔	
↓	↓	H	⊓	⊔	One-shot
L	X	↑	⊓	⊔	outputs
X	L	↑	⊓	⊔	

A constant low input at A1, A2 or B will hold the 74121 monostable multivibrator in its stable condition — with a low Q output and a high $\overline{Q}$ output. High inputs at A1 and A2 have the same effect.

There are five input signal combinations that will generate one-shot outputs. These input signal combinations are identified in the function table above.

With regard to the function table, symbols are used as follows:

X   represents a "don't care"

↓   represents a one-to-zero logic transition

↑   represents a zero-to-one transition

⊓   represents a one-shot with a zero monostable logic level and a one pulse level

⊔   is the NOT of ⊓

The duration of the one-shot output is determined by a resistor-capacitor network, just as described for the Signetics 555 monostable multivibrator; but there are some differences. The 74121 provides an internal resistor which may be accessed by connecting R_{INT} (pin 9) to V_{CC} (pin 14). A variable external resistor may be connected between R_{INT} (pin 9) or R_{EXT} (pin 11) and V_{CC} (pin 14).

An external timing capacitor, if present, will be connected between C_{EXT} (pin 10) and R_{EXT} (pin 11).

Here is one way in which a 74121 monostable multivibrator may be connected:
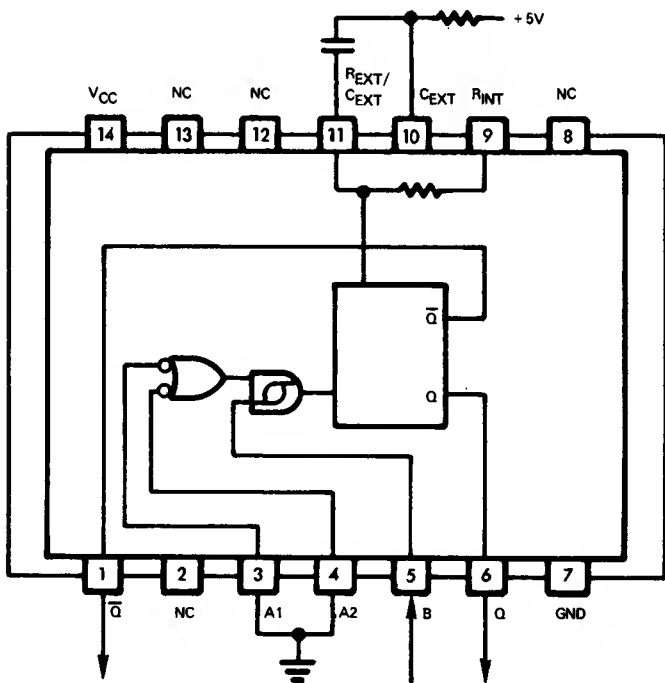
This use of the 74121 monostable multivibrator corresponds to the bottom two lines of the function table.

An external resistor/capacitor network controls one-shot pulse duration. Each one-shot pulse will be triggered by a low-to-high transition at pin 5 (B).

From the programming point of view, there are only two significant features of the 74121 monostable multivibrator:

1) **The monostable outputs are equivalent to binary digits of fixed value.** Any Immediate instruction which loads a zero or a one into any register bit simulates the monostable output. Here is an example:

                    LDA B   #4   SET BIT 3 OF ACCUMULATOR B TO 1. RESET ALL
                                              OTHER BITS

Bit 3 of Accumulator B is equivalent to a flip-flop; so is every other bit of Accumulator B, and every other Accumulator.

2) **A one-shot output becomes a time delay of fixed value.** We will show how this time delay may be computed within a microcomputer system but first let us examine the 74107 master-slave flip-flop.

THE 74107 DUAL J-K MASTER-SLAVE FLIP-FLOP WITH CLEAR

Consider the 74107 master-slave flip-flop. This flip-flop is illustrated as follows:

INPUTS				OUTPUTS	
1 CLR or 2CLR	1CK or 2CK	1J or 2J	1K or 2K	1Q or 2Q	$1\overline{Q}$ or $2\overline{Q}$
L	X	X	X	L	H
H	⎍	L	L	Stay as you were	
H	⎍	H	L	H	L
H	⎍	L	H	L	H
H	⎍	H	H	Change state regardless of previous state	
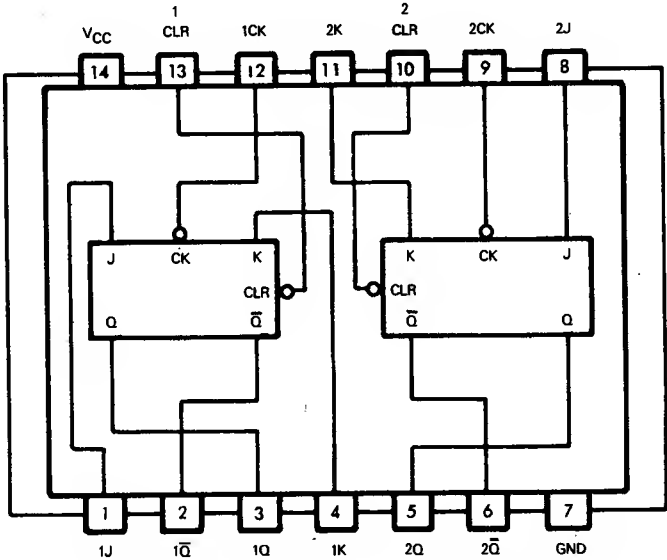
⎍   identifies a clock pulse; the way in which it is used is described below.

X   means "don't care"

Let us examine the function table illustrated above. Unless you are familiar with this type of logic device, its features are not self-evident.

The connotation "master-slave" identifies a circuit which is, in fact, two flip-flops. Therefore, there are four flip-flops in the 74107 device illustrated above.

MASTER-SLAVE FLIP-FLOPS

The flip-flops in each master-slave pair respond to a clock signal as follows:

CLOCK

Isolate the slave flip-flop
from the master

Connect master and slave
flip-flops, thus creating
output signals

Master flip-flop accepts
input signals

Isolate the master flip-flop
from the input signals
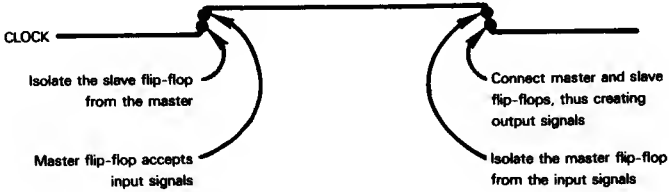
The significance of this clock signal response is that the flip-flop inputs must be present at the positive edge of the clock signal; these inputs must remain steady while the clock signal is high. The flip-flop outputs, however, do not change state until the negative edge of the clock signal.

The Clock signal may be used to create time delays. The 74107 flip-flop output is determined by input signal levels as they existed some time period earlier. This may be illustrated as follows:

CLOCK

Condition of
J and K here

Determine condition
of Q and Q̄ here

Here is a specific example:

① ② ③ ④ ⑤ ⑥

CLOCK

J

K

Q
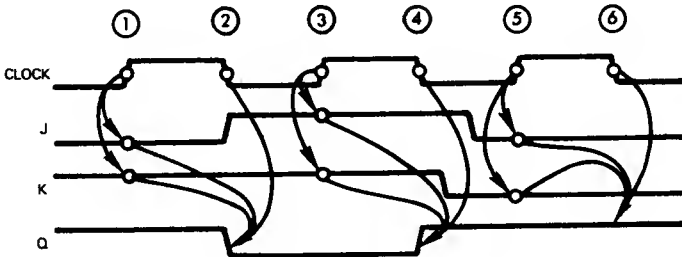
The following description of the timing diagram illustrated above is keyed to the circled numbers above the clock signal.

At ②, the Q output goes low, because at ① J was low and K was high.

At ④, Q changes state because at ③ J and K were both high.

At ⑥, Q remains unaltered because at ⑤ J and K were both low.

2-35

MICROCOMPUTER SIMULATION OF REAL TIME

What is the significance of the 555 monostable multivibrator and the master-slave flip-flops? When it comes to microcomputer simulation of these devices, there is only one feature that is important to our present discussion — and that is the concept of real time.

The 555 monostable multivibrator creates high logic level pulses at its output, where the duration of the high logic level is a controllable real time function.

The 74107 master-slave flip-flop allows an output signal to be generated based on input conditions as they existed some real time earlier.

MICROCOMPUTER TIMING INSTRUCTION LOOPS

It is simple enough to create a time delay using a microcomputer system — providing the microcomputer system is not being called upon to perform any other simultaneous operations. Consider the following instruction sequence:

<div style="border:1px solid">TIMING
SHORT TIME
INTERVALS</div>

Cycles				
		LDA A	#TIME	LOAD TIME CONSTANT INTO ACCUMULATOR A
2	LOOP	DEC A		DECREMENT ACCUMULATOR A
4		BNE	LOOP	REDECREMENT IF NOT ZERO

The above instruction sequence loads a data value, represented by the label #TIME, into Accumulator A. The Accumulator is decremented until it reaches zero, at which time program execution continues. Let us assume that a one microsecond clock is being used by the microcomputer system. The DEC and BNE instructions, taken together, execute in 6 cycles — which is equivalent to 6 microseconds. This means that the program sequence illustrated above can cause a delay with a minimum value of 6 microseconds (when #TIME equals 1), increasing in 6 microsecond steps to a maximum delay of 1536 microseconds, which is equivalent to 6 x 256. This maximum time delay will result when #TIME has an initial value of zero, since #TIME is decremented BEFORE being tested to see if it is zero; therefore the time out occurs when 1 decrements to 0, not when 0 decrements to FF_{16}.

The MC6800 instruction set is heavily oriented toward memory reference instructions; this being the case, **you should always examine the use of read/write memory, instead of an Accumulator, when using the MC6800.** Thus the time delay instruction sequence could be rewritten as follows:

<div style="border:1px solid">MEMORY
REFERENCE
INSTRUCTIONS</div>

Cycles				
		LDA A	#TIME	LOAD TIME CONSTANT INTO ACCUMULATOR A
		STA A	TLOC	STORE IN TIME CONSTANT LOCATION
6	LOOP	DEC	TLOC	DECREMENT CONSTANT LOCATION CONTENTS
4		BNE	LOOP	REDECREMENT IF NOT ZERO

The first point to note regarding the memory reference time delay loop illustrated above is that it requires ten machine cycles in order to execute the loop once; again assuming a one microsecond clock time, delays ranging between 10 and 2,560 microseconds may be created in increments of 10 microseconds.

On first inspection the memory reference time delay loop illustrated above may seem to have only disadvantages as compared to the use of an Accumulator to hold the time delay constant. But the memory reference instruction loop may have advantages. We show two instructions immediately preceding the time delay loop loading a time constant represented by the label #TIME into a memory location represented by the label TLOC. In a real program, the data value represented by the label #TIME might be loaded into the location represented by TLOC during an early initialization phase of program logic. Subsequently the time delay loop could be executed without disturbing the contents of the Accumulator. For example, an interrupt may trigger the execution of the time delay loop. Using the memory reference version of the time delay program, it would not be necessary to save and restore the Accumulator contents since the time delay loop instruction sequence does not modify the contents of any CPU register.

Longer time delays can be generated by having a 16-bit counter. The Index register is a very convenient means of implementing a timing loop that uses a 16-bit counter; this may be illustrated as follows:

```
Cycles
                LDX     #TIM16    LOAD TIME CONSTANT INTO INDEX REGISTER
    4    LOOP   DEX               DECREMENT INDEX REGISTER
    4           BNE     LOOP      REDECREMENT IF NOT ZERO
```

Eight machine cycles are required in order to execute the decrement loop once. Thus, time delays ranging between 8 microseconds and 0.524288 seconds may be generated in increments of 8 microseconds. Again we assume a 1 microsecond clock. The maximum time delay is computed when 0 is initially loaded into the Index register. The maximum time delay is then computed as follows:

$$65,536 \times 8 = 524,288 \text{ microseconds}$$

It is a little more complicated creating a long time delay instruction sequence where the 16-bit timer constant must be stored in two memory locations. Here is the appropriate instruction sequence:

```
Cycles
                LDA  A    #TIMHI     LOAD INITIAL 16-BIT TIME CONSTANT
                STA  A    TLOC       INTO TWO CONTIGUOUS MEMORY
                LDA  A    #TIMLO     LOCATIONS ADDRESSED BY TLOC AND
                STA  A    TLOC + 1   TLOC + 1
    6    LOOP   DEC       TLOC + 1   DECREMENT LOW ORDER BYTE OF COUNTER
    4           BNE       LOOP       REDECREMENT IF NOT ZERO        .
    6           DEC       TLOC       DECREMENT HIGH ORDER BYTE OF COUNTER
    4           BNE       LOOP       REDECREMENT IF NOT ZERO
```

Logic within the instruction loop illustrated above decrements the low order eight bits of the 16-bit counter; each time the low order eight-bit decrement creates a 0 value, the high order eight bits are decremented. When the high order eight bits decrement to 0, logic exits from the time delay loop. When the high order eight bits of the counter decrement to a nonzero value, then the low order eight bits must be decremented again through a full decrement cycle before redecrementing the high order eight bits.

Now the actual simulation of a one-shot is complicated by the fact that we may compute time delays, but when does the time delay begin? For digital logic devices the answer is simple: the time delay begins when an input signal changes state:

TIME DELAY INITIATION

To parallel this concept within a microcomputer program, we must initiate a time delay upon completing some other program sequence's execution. This concept may be illustrated as follows:

```
        -
        -
        -
    ·  JMP     DELY      LAST INSTRUCTION OF SOME PRIOR SEQUENCE
        -
        -
        -
DELY   LDA  A  #TIME     SHORT TIME INTERVAL INSTRUCTION
LOOP   DEC  A            SEQUENCE
       BNE     LOOP
```

There is another problem associated with creating time delays within a microcomputer system by executing instruction loops, as we have described: the microcomputer is, in essence, doing no useful work during the time delay. There may be a simple remedy to this problem, providing we can define a program for the microcomputer to execute during the period of the time delay. This may be illustrated as follows:

| Start of desired time delay | An instruction sequence whose execution time is known exactly executes during this time period. This is a coarse time interval. | The remaining time is timed out using a time delay instruction loop. This is a fine tuning time interval. |
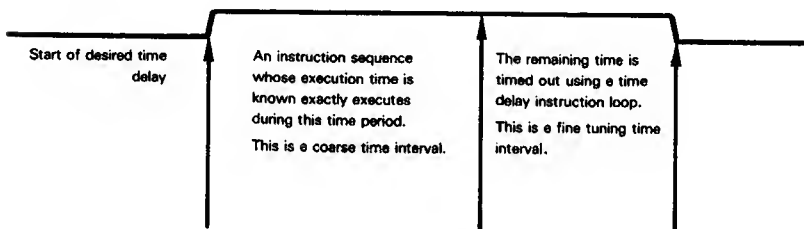
We must assume that we can calculate the exact time it will take for our program to execute within the one-shot time delay; also, the computed time must be less than, or equal to the time delay. Not many programs are going to fit this description. If, for example, more than one instruction sequence may get executed, depending on current conditions, then there may be many different times required for a program to execute. Still, so long as there are a fixed number of identifiable branches, the problem is tractable and may be illustrated as follows:

Start of desired time delay

◇ represents decision and branch logic

─── represents program execution time

- - - represents time delay instruction loop execution time
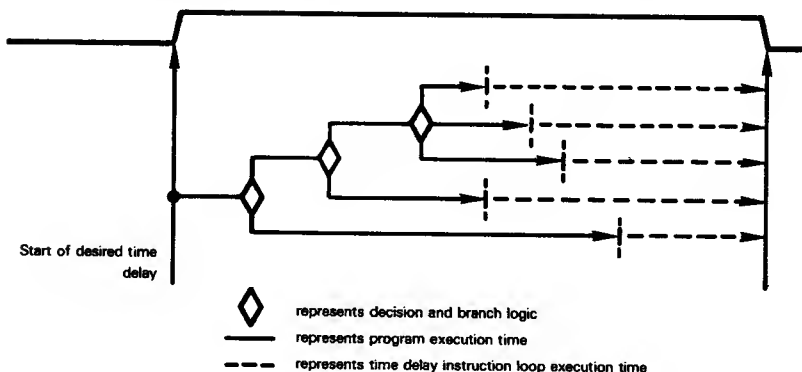
Now each "limb" of the program branches will end as follows:

```
        LDA  A    DLY1     LOAD FIRST TIME DELAY
        JMP       LOOP     START TIME DELAY LOOP
        -
        -
        -
        LDA  A    DLY2     LOAD SECOND TIME DELAY
        JMP       LOOP     START TIME DELAY LOOP
        -
        -
        -
        LDA  A    DLY3     LOAD THIRD TIME DELAY
        JMP       LOOP     START TIME DELAY LOOP
        -
        -
        -
        LDA  A    DLY4     LOAD FOURTH TIME DELAY
        JMP       LOOP     START TIME DELAY LOOP
        -
        -
        -
        LDA  A    DLY5     LOAD FIFTH TIME DELAY
        JMP       LOOP     START TIME DELAY LOOP
        -
        -
        -
LOOP    DEC  A             SHORT TIME INTERVAL INSTRUCTION
        BNE       LOOP     SEQUENCE
```

It is more common than not for a microcomputer program to contain numerous conditional branches; there may be hundreds of different possible execution times depending on various combinations of current conditions. Executing a program within the time interval of the required delay now becomes impractical, because the logic needed to compute remaining time for the innumerable program branches is just too complicated.

THE LIMITS OF DIGITAL LOGIC SIMULATION

An MC6800 microcomputer can compute time delays so long as no other program needs to be executed during the time delay, or providing a very simple instruction sequence with very limited branching is executed during the time delay.

You cannot simulate simultaneous time delays, nor can you simulate a time delay which must occur in parallel to un-definable parallel program executions. External logic must handle all such time delays.

SIMULTANEOUS TIME DELAYS

INTERFACING WITH EXTERNAL ONE-SHOTS

Note that even though external logic may have to create time delays, it is very easy for the microcomputer system to trigger the start of the time delay and for the external logic to report the completion of the time delay.

We can identify the start of a time delay by simply outputting an appropriate binary digit. Look again at the way "Signal Out" was output to external logic by the signal inverter simulation. Outputting a signal to external logic is indeed very easy. Consider the following four instructions:

LDA A	A	#0	LOAD A 0 INTO ACCUMULATOR A
STA A	A	PORTB	OUTPUT VIA I/O PORT B
LDA A	A	#2	LOAD A 1 INTO ACCUMULATOR A BIT 1
STA A	A	PORTB	OUTPUT VIA I/O PORT B

A 1 is output at pin 1 of I/O Port B. Assuming that the pin associated with this I/O port is connected to the trigger of a multivibrator, and that this connection was previously high, then the simple execution of the above instructions will trigger a one-shot.

This may be illustrated as follows:
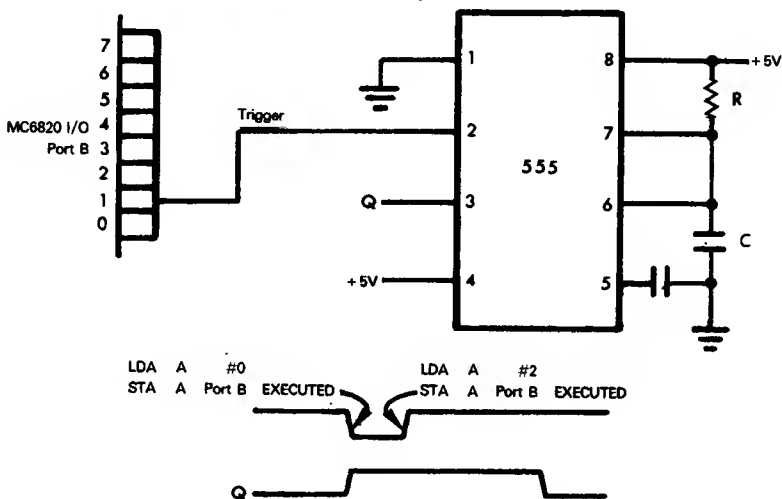
It is equally easy for external logic to signal the end of a time delay.

If we are dealing with "greater than or equal to" logic, all that is necessary is for the one-shot output to be connected to another pin of a microcomputer I/O port:
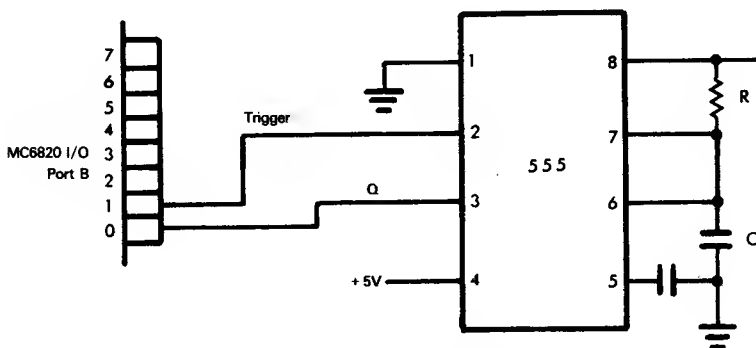
Signals arriving at pins of I/O ports are buffered. The program being executed by the microcomputer may, at any time, input the contents of the I/O port and test the condition of bit 0, which has been wired to the Q output. When this bit is found to equal 0, microcomputer program logic knows that the time interval has been surpassed.

The following instruction sequence will test the I/O port and clear the "time interval complete" status being reported by I/O Port B, pin 0:

```
              LDA A    PORTB    INPUT CONTENTS OF I/O PORT B TO ACCUMULATOR A
              AND A    #1       MASK OUT ALL BITS BAR BIT 0
              BNE      NEXT     CONTINUE IF BIT IS 1
TIME OUT PROGRAM BEGINS HERE
                 -
                 -
                 -
NEXT                           TIME NOT OUT PROGRAM BEGINS HERE
```

The LDA instruction moves the current contents of I/O Port B to Accumulator A.

The following AND instruction sets all Accumulator bits to 0 bar the bit corresponding to I/O Port B, pin 0:

```
              7 6 5 4 3 2 1 0 ◄─Bit No.
              X X X X X X X Y   Accumulator Contents
              0 0 0 0 0 0 0 1   Hexadecimal 01
              0 0 0 0 0 0 0 Y   Result of AND
```

If the binary digit input from pin 0 of I/O Port B is 1, then the Q output is still high. The BNE NEXT instruction simply continues program execution.

If bit 0 of I/O Port B is 0, then the time delay is over; we branch to a program sequence which only gets executed immediately following a time out.
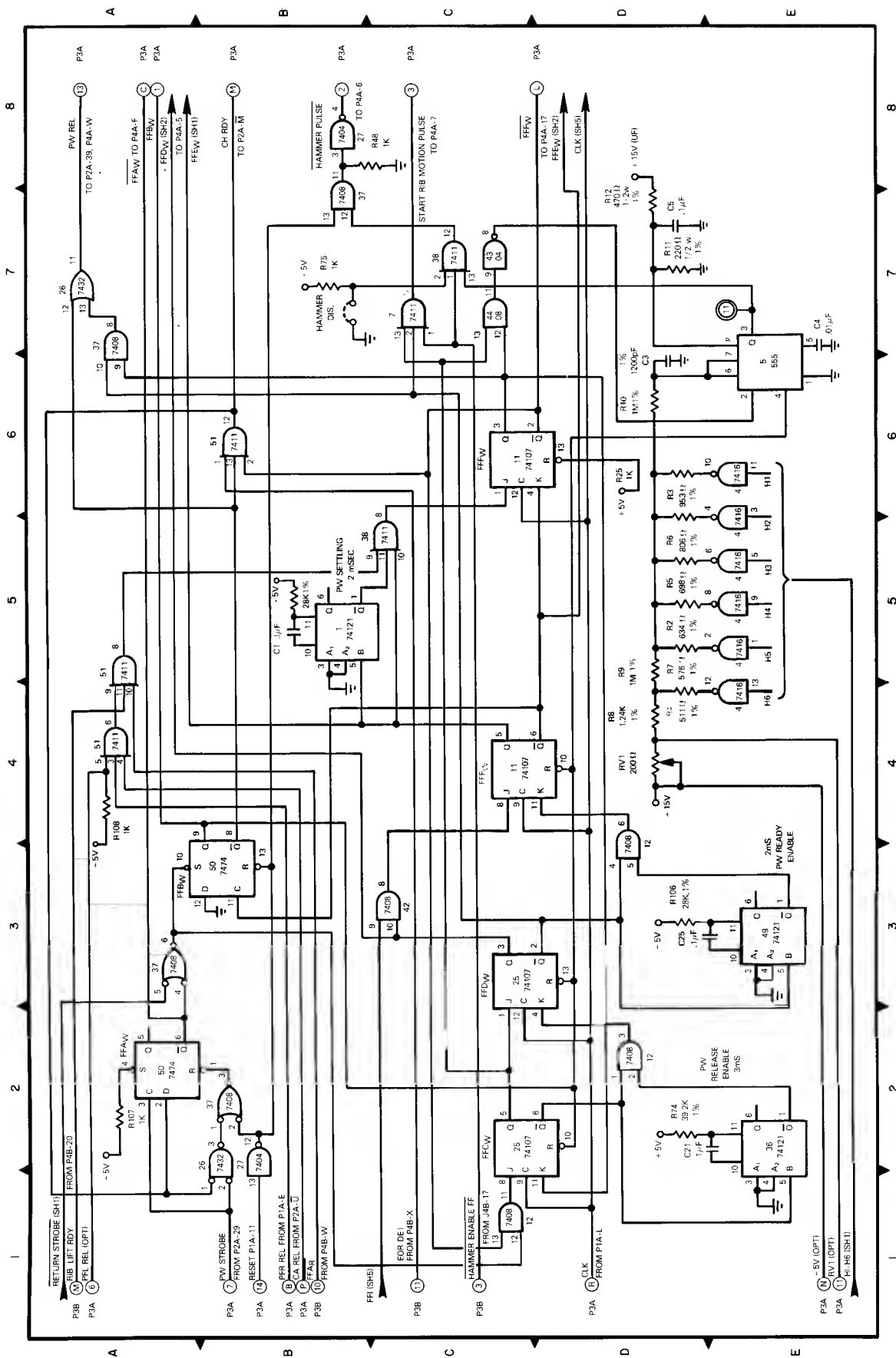
Figure 3-1. Printwheel Control Logic

Chapter 3
A DIRECT DIGITAL LOGIC
SIMULATION

The discrete logic devices which we simulated in Chapter 2 were not selected at random; correctly sequenced, they will simulate the logic illustrated in Figure 3-1. This logic is a portion of the printer interface for the Qume Q-Series and Sprint Series printers. Figure 3-2 is the timing diagram that goes with Figure 3-1. We are going to describe both figures at a very elementary level.

Now the purpose of this chapter is to provide a one-for-one correlation between microcomputer assembly language programming and digital logic design. What you must understand is that while such a one-for-one correlation can be forced, it is not natural; and that is where the problem in understanding lies. Microcomputer programs should be written to stress the nature of microcomputers, not the characteristics of digital logic.

The correct way to program a microcomputer is described beginning at Chapter 4.

Nevertheless, the juxtaposition of digital logic design and microcomputer programming is underscored in this chapter. This is the chapter that bridges two concepts; and for that reason it is the most important chapter in this book. If you are a logic designer, this chapter is important because it will eliminate digital logic concepts which are inapplicable to microcomputers. If you are a programmer, this chapter is important because it will acquaint you with a new programming goal — efficient logic implementation.

To achieve the goal of this chapter, we will describe the logic illustrated in Figures 3-1 and 3-2; the description will be careful and detailed, so that you can follow this chapter, even if you are not a logic designer. As the logic description proceeds, we will blend in assembly language — in easy stages.

If you understand digital logic, it is particularly important that you confine your reading to the bold face type in this chapter. The logic of Figure 3-1 has been described in sufficient detail to meet the needs of a programmer, or a reader, with no logic background.

HOW THE QUME PRINTER WORKS

The active Qume printing element is a 96-petal printwheel, with one character on each petal:

COURTESY OF QUME CORPORATION

A character is printed by moving the printwheel until the appropriate petal is in front of a solenoid driven printhammer. The printhammer is then fired; it strikes the printwheel petal, which marks the paper:
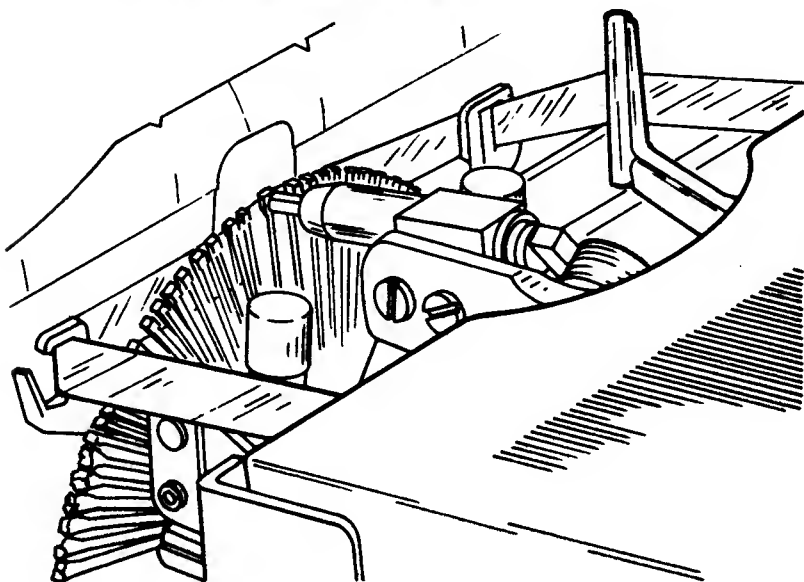
Whenever a character is not in the process of being printed, the printwheel is positioned with a short petal immediately vertical, so that the character just printed is visible:
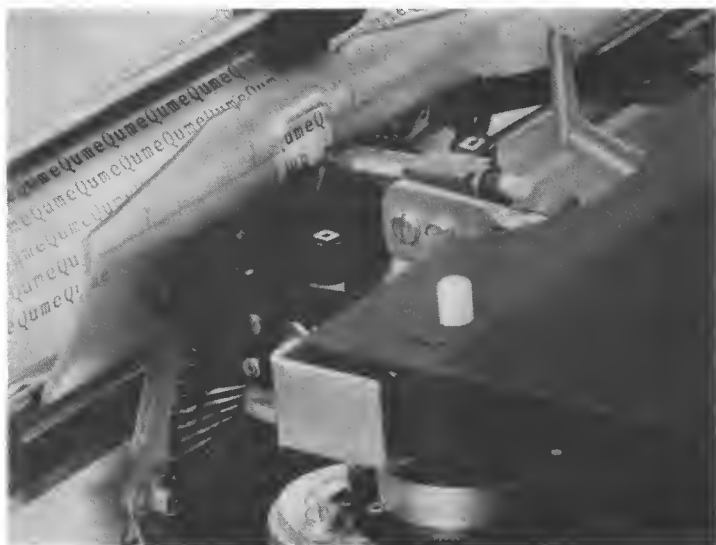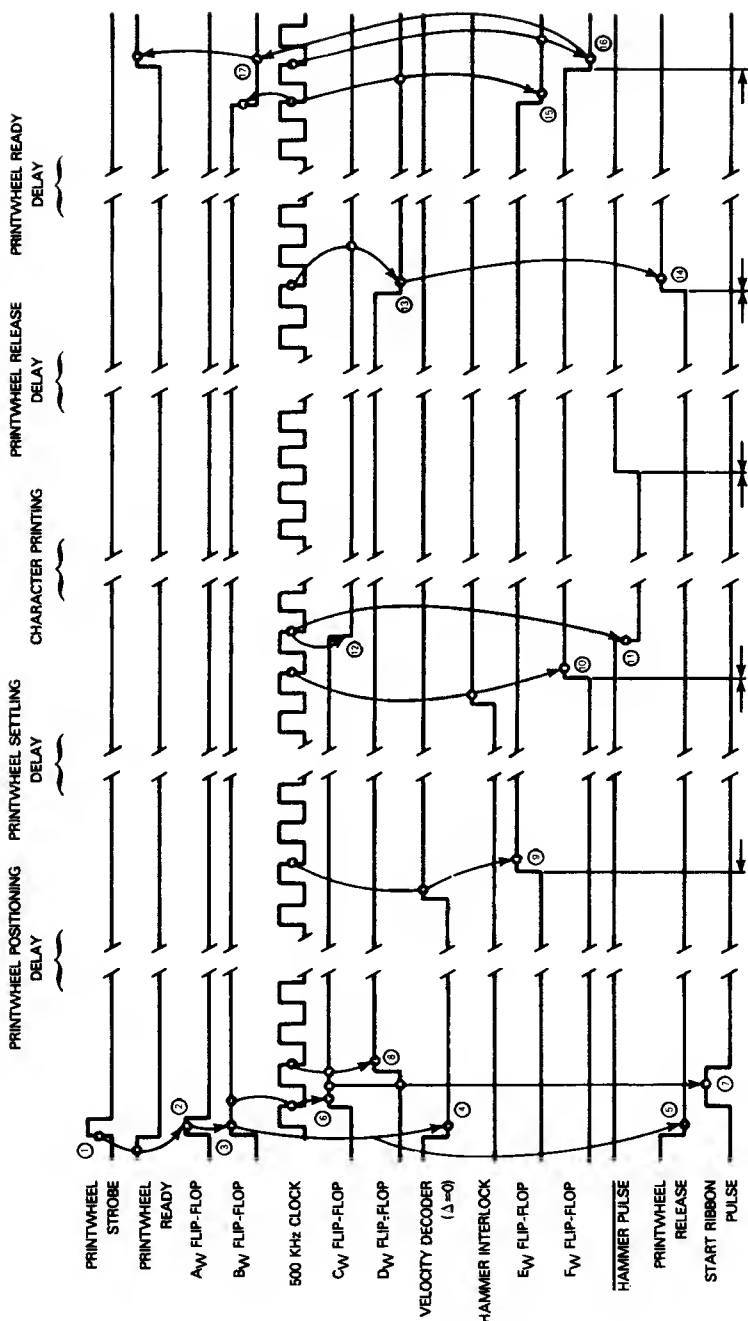
Figure 3-2. Printwheel Control Logic Timing Diagram

As part of the print cycle, the printer ribbon and paper carriage must be moved.

Every character is printed according to a definite sequence of events, collectively referred to as a "print cycle". The logic illustrated in Figure 3-1 controls the character print cycle. **These are the events which must occur within a print cycle:**

1) First, the print cycle must be initiated. A **signal (PW STROBE) is pulsed high to initiate the print cycle:**

PW STROBE

2) The print cycle will endure for a fixed time interval. Obviously during this time interval another print cycle must not be initiated. Therefore the **external logic** responsible for generating PW STROBE true **must be given a signal identifying the duration of the print cycle. This signal is PRINTWHEEL READY, also called CH RDY:**

PRINTWHEEL READY
CH RDY
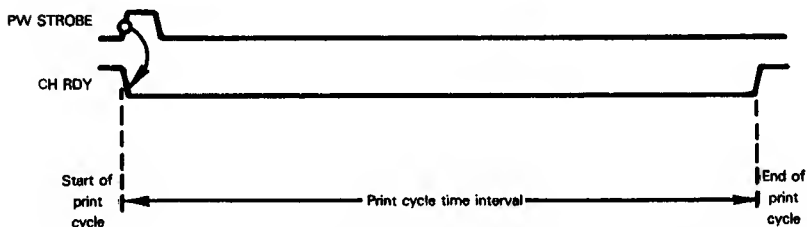
The sequence of events which actually cause a character to be printed can now proceed with the assurance that external logic will not attempt to start printing the next character before the current print cycle has gone to completion.

3) **The printwheel is moved from its position of visibility until the appropriate character petal is in front of the printhammer:**
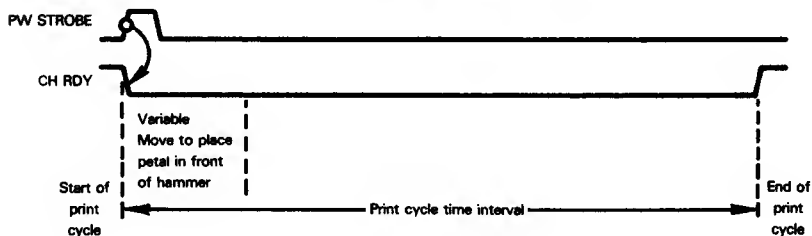
A variable time delay is needed by the printwheel positioning logic. Obviously it will take longer to position a petal that is far from the position of visiblity than to position to an adjacent petal.

4) Before the printhammer is fired, **the printwheel must be given time to settle.** A fixed. 2 millisecond time delay is sufficient:

PW STROBE

CH RDY

Variable
Move to place
petal in front
of hammer

Fixed
Printwheel
settling
time

Start of
print
cycle

Print cycle time interval

End of
print
cycle
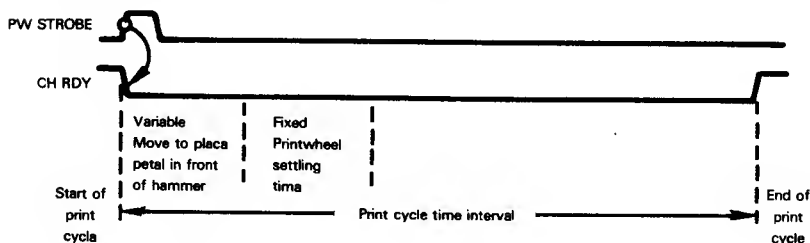
Settling time delays are a very important aspect of the logic supporting any type of mechanical movement. It is easy to draw a clean line showing movement velocity as follows:

SETTLING DELAYS

Accelerate

Move

Decelerate

Stop

Stop
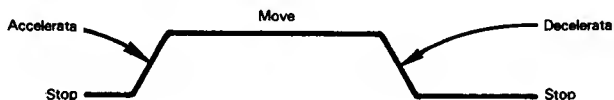
But in reality, movement occurs like this:

Accelerate

Move

Decelerate

Stop

Stop
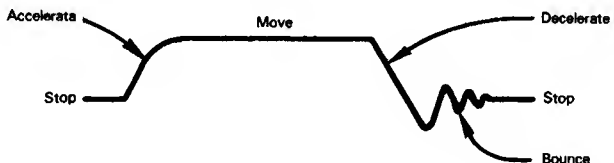
Bounce

The bounce that follows deceleration must be passed over by a settling time delay.

A blurred character will be printed if the printwheel is still vibrating when the printhammer hits a petal against the paper.

5) At the end of the printwheel settling time delay, **the printhammer can be fired.** This is done by outputting an impulse to a solenoid. **Six firing impulse intensities are provided,** since some characters have a more substantial surface area than others. To strike a comparatively large surface area like a "W" with the same intensity that you strike a small

character, like a ".", would produce unevenness in the density of the printed text. The duration of the printhammer solenoid pulse is controlled by the next time delay:
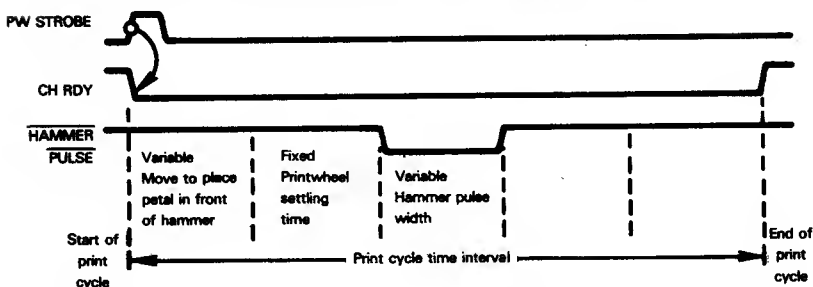
The bar over $\overline{\text{HAMMER PULSE}}$ identifies the signal as one which is low when active.

6) At the completion of the printhammer pulse time delay, the hammer has struck a petal and forced it onto the paper. Now **the hammer must be given time to return to its prefiring position.** A 3 millisecond delay is generated for this purpose:
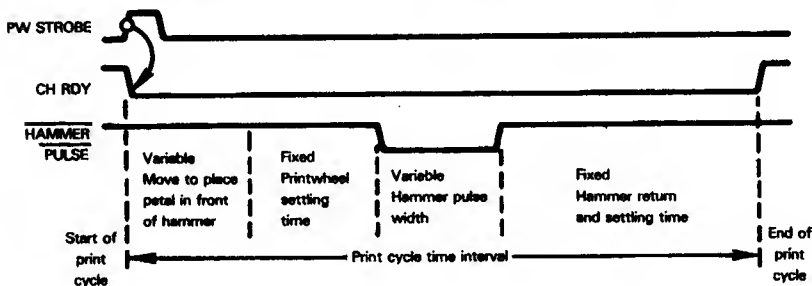
7) Now **the printwheel can be moved to its position of visibility** and the paper carriage can be advanced to the next character position. The printwheel's "position of visibility" is its normal inactive position; in this position a short petal is in front of the

> **PRINTWHEEL POSITION OF VISIBILITY**

printhammer, so the most recently printed character is visible above the short petal; hence the "position of visibility". Had we not given time for the printhammer to settle back before moving the printwheel to its position of visibility, a printwheel petal may have been broken, striking the tip of the still protruding hammer. Also the paper may have smudged moving against a bent petal. Since the printhammer has been given time to fully retract, none of these problems will arise.

A final 2 millisecond time delay allows the printwheel and paper carriage to reposition themselves:

PW STROBE

CH RDY

HAMMER PULSE

| Variable Move to place petal in front of hammer | Fixed Printwheel settling time | Variable Hammer pulse width | Fixed Hammer return and settling time | Fixed Final movements delay |

Start of print cycle |◄──────── Print cycle time interval ────────►| End of print cycle
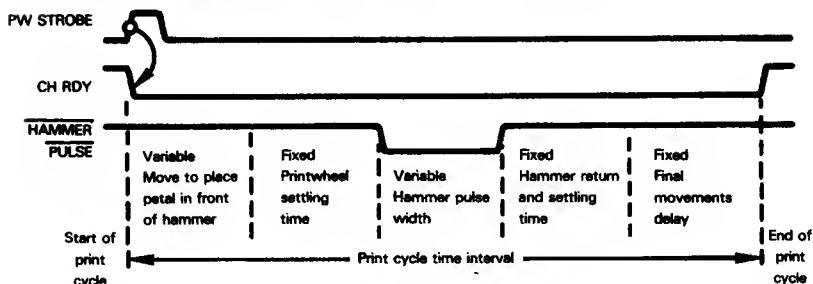
8) What about ribbon logic? **In order to get a clean impression on the paper, a fresh piece of ribbon must present itself between the character petal and the paper.** Shortly after the beginning of the print cycle, therefore, a signal (START RIBBON MOTION PULSE) is output to external logic, which actually controls ribbon movement. This external logic (it is not part of Figure 3-1) sends back

```
┌──────────┐
│ START    │
│ RIBBON   │
│ PULSE    │
├──────────┤
│ FFA      │
└──────────┘
```

a ribbon movement completed signal (FFA) since we cannot allow the printhammer to be fired while the ribbon is still moving. Thus **the ribbon is advanced while the printwheel is initially being positioned and settled:**

PW STROBE

CH RDY

HAMMER PULSE

| Variable Move to place petal in front of hammer | Fixed Printwheel settling time | Variable Hammer pulse width | Fixed Hammer return and settling time | Fixed Final movements delay |

Start of print cycle |◄──────── Print cycle time interval ────────►| End of print cycle

Start ribbon movement

Ribbon movement complete
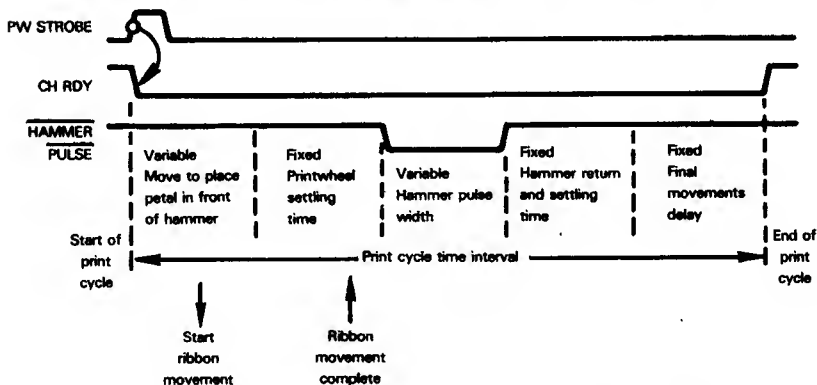
In summary, a print cycle consists of five time delays, each time delay starts out with a flurry of logical activity, followed by a period of mechanical movement.

INPUT AND OUTPUT SIGNALS

Now that you have a general understanding of the functions which are controlled by logic in Figure 3-1, **the next step is to take a closer look at input and output signals.**

In order to know what to do, and when to do it, we must rely entirely upon input signals. Similarly, output signals represent the only way in which we can transmit control information to external logic.

Our limited goal, at this point, is to understand what function each input and output signal performs, and how — physically — we are going to handle the signals. We will discuss the "how" first.

INPUT/OUTPUT DEVICES

Tha principal device used to transmit signals and data between an MC6800 microcomputar system and external logic is tha MC6820 Paripheral Interfaca Adaptar (PIA).

Wa are going to usa two MC6820 Peripheral Interface Adapters.

PERIPHERAL
INTERFACE
ADAPTER

LATCHED
BUFFER

Since this device has been described in "An Introduction To Microcomputers", we are going to assume that you understand its capabilities and organization superficially; if you do not, see "An Introduction To Microcomputers; Volume II — Some Real Products" before continuing, otherwise you will not understand the discussion which follows.

THE MC6820 PERIPHERAL INTERFACE ADAPTER (PIA)

Tha MC6820 Paripheral Interfaca Adaptar (PIA) provides 16 I/O pins which may be grouped into I/O ports as follows:
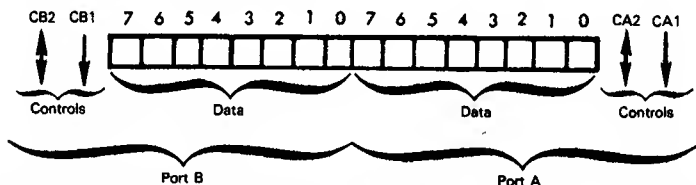
Each port has two associated control signals. CA1/CB1 are input control signals; that is to say, external logic inputs control information to the MC6820 via CA1/CB1. CA2/CB2 are bidirectional control signals.

Pins of I/O Ports A and B may be assigned individually to input data or output data; but no pin can support bidirectional data transfars. Data transfers may occur unaccompanied by any handshaking controls, or with handshaking controls.

I/O PORT
MODES

Control signals CA1 and CA2 can be used with I/O Port A to generate input data transfer with handshaking.

Control signals CB1 and CB2 can be used with I/O Port B to generate output data transfer with handshaking.

"Input" refers to data transfer from external logic to the MC6820; "output" refers to data transfer from the MC6820 to external logic.

Table 3-1 summarizes the operating modes available with MC6820 I/O ports.

Table 3-1. MC6820 Operating Modes

OPERATING MODE	MC6800 AVAILABILITY
Simple input without handshaking	I/O Port A or B
Simple output without handshaking	I/O Port A or B
Bidirectional I/O without handshaking	Not available, but individual pins of either I/O port may be separately assigned to input or output
Input with handshaking	I/O Port A only
Output with handshaking	I/O Port B only
Bidirectional I/O with handshaking	Not Available

Six individually addressable locations are present within an MC6820; there are the two I/O ports (A and B), a Data Direction register associated with each I/O port, and a Control register associated with each I/O port. These six addressable locations may be illustrated as follows:

MC6820 ADDRESSABLE LOCATIONS
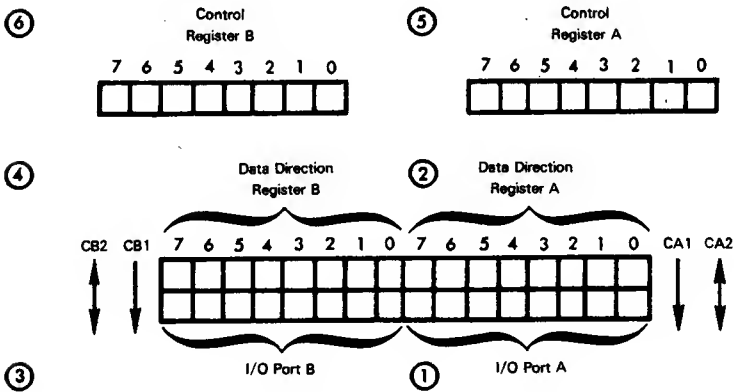
The six addressable locations are identified by numbers ① through ⑥

I/O Ports A and B are the conduits via which actual data transfers occur. The two I/O ports are accessed as two individually addressable memory locations.

Associated with each I/O port is a Data Direction register. The Data Direction register identifies each pin of its associated I/O port as being dedicated to either input or output. These are write-only registers. You must write a control word into each Data Direction register; a 0 in a bit position configures the corresponding I/O port as an input, while a 1 results in an output.

I/O Port Pins
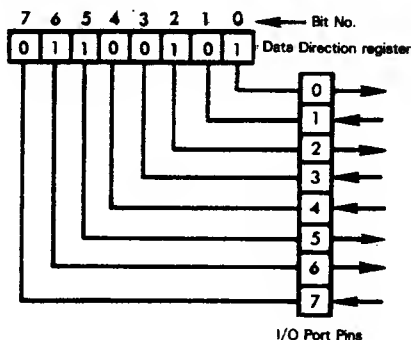
I/O Ports A and B will both be configured as 8-bit input ports when the MC6820 is reset, since RESET clears all internal registers.

The two Data Direction registers constitute two additional addressable memory locations.

Associated with each I/O port there is also a Control register. Each Control register constitutes an additional addressable memory location, and the contents of the Control register determine the manner in which its associated I/O port and control signals will operate.

But an MC6820 has assigned to it just four memory addresses.
These four addresses are created by the way in which select logic connects five MC6820 pins to the Address Bus. These are the five select signals, and their function:

	MC6820
	REGISTER
	ADDRESSING

CS0   Must be high
CS1   Must be high
$\overline{CS2}$ Must be low
RS0 ⎱ Select one of four addressable locations
RS1 ⎰ within the MC6820

We will use two MC6820 devices. The select lines for these two devices will be connected to the Address Bus as follows:
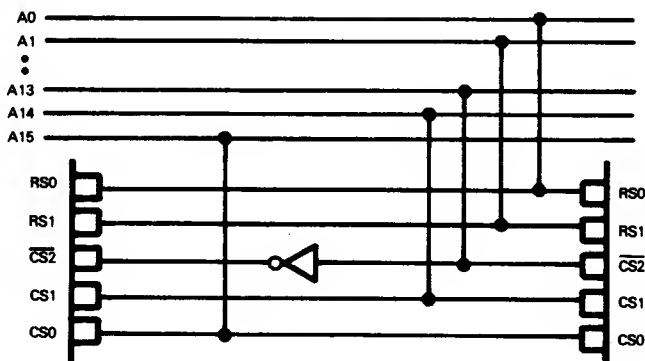
As a consequence of the above connections, the MC6820 PIAs will respond to the following memory addresses:
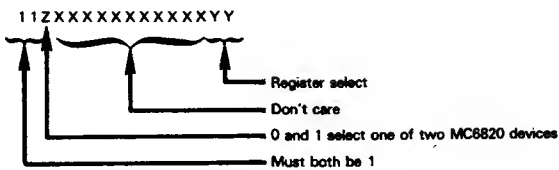
Thus, all addresses in the range C000$_{16}$ through DFFF$_{16}$ will select one MC6820; addresses in the range E000$_{16}$ through FFFF$_{16}$ will select the other MC6820, even though in each case only four addresses are needed. **We will use the four addresses C000$_{16}$, C001$_{16}$, C002$_{16}$ and C003$_{16}$, and E000$_{16}$, E001$_{16}$, E002$_{16}$ and E003$_{16}$;** these addresses are generated if all X digits are assumed equal to 0.

Each MC6820 interprets its four memory addresses as follows:

Lowest address: (e.g., C000$_{16}$)	Data Direction Register A or I/O Port A
Next address: (e.g., C001$_{16}$)	Data Direction Register B or I/O Port B
Next address: (e.g., C002$_{16}$)	Control Register A
Highest address: (e.g., C003$_{16}$)	Control Register B

As illustrated above, a single memory address is assigned to access either a Data Direction register or its associated I/O port. Which of the two will in fact be accessed, depends on the contents of the associated Control register bit 2. Thus, Table 3-2 summarizes MC6820 register addressing.

Table 3-2. Addressing MC6820 Internal Registers

To illustrate MC6820 addressing, suppose again that the four addresses $C000_{16}$, $C001_{16}$, $C002_{16}$ and $C003_{16}$ select an MC6820. This is how addressable locations within the MC6820 would actually be selected:

Address	Selected
$C000_{16}$	I/O Port A Data Direction register, if $C002_{16}$, bit 2 = 0
	I/O Port A Data buffer, if $C002_{16}$, bit 2 = 1
$C001_{16}$	I/O Port B Data Direction register, if $C003_{16}$, bit 2 = 0
	I/O Port B Data buffer, if $C003_{16}$, bit 2 = 1
$C002_{16}$	I/O Port A Control register
$C003_{16}$	I/O Port B Control register

When would you select the Data Direction register, rather than the associated I/O port?

In the normal course of events, you will want to address an I/O port rather than its Data Direction register. This being the case, bit 2 of the associated Control register will normally contain 1. Thus, **you will use the following instruction sequence to access a Data Direction register:**

<div style="float:right; border:2px solid black; padding:4px;">
MC6820 DATA DIRECTION REGISTER ADDRESSING
</div>

LDA A	CX	LOAD THE CONTENTS OF CONTROL REGISTER X INTO ACCUMULATOR A
TAB		SAVE IN ACCUMULATOR B
AND A	#$FB	SET BIT 2 TO 0
STA A	CX	RESTORE CONTENTS OF CONTROL REGISTER X WITH BIT 2 = 0
LDA A	MASK	LOAD DATA DIRECTION REGISTER WITH AN APPROPRIATE
STA A	CX-2	MASK TO SET I/O PORT PIN DEFINITIONS
STA B	CX	RESTORE ORIGINAL CONTROL REGISTER X CONTENTS

Let us examine the above instruction sequence. Control Register X is associated with some I/O Port X. We assume that under normal circumstances bit 2 of the Control register is set to 1, since under normal circumstances you will want to access an I/O port rather than its Data Direction register. Therefore we must reset bit 2 of the Control register to 0 before accessing the Data Direction register. This is done by reading the Control register contents into Accumulator A, masking it with a mask that contains 0 in bit position 2 and 1s in all other bit positions, then outputting the result back to the Control Register CX. The initial Control register contents are saved in Accumulator B so that they can be restored later. This logic may be illustrated as follows:

		Control Register X	Data Direction Register X	Accumulator A	Accumulator B
Initial Contents		X X X X X 1 X X	?	?	?
LDA A	CX	X X X X X 1 X X	?	X X X X X 1 X X	
TAB		X X X X X 1 X X	?	X X X X X 1 X X	X X X X X 1 X X
AND A	#$FB	X X X X X 1 X X	?	1 1 1 1 1 0 1 1 X X X X X 0 X X	X X X X X 1 X X
STA A	CX	X X X X X 0 X X	?	X X X X X 0 X X	X X X X X 1 X X
LDA A	MASK	X X X X X 0 X X	?	MASK	X X X X X 1 X X
STA A	CX-2	X X X X X 0 X X	MASK	MASK	X X X X X 1 X X
STA B	CX	X X X X X 1 X X	MASK	MASK	X X X X X 1 X X

When the contents of Control Register CX bit 2 are 0, we can access Data Direction Register X. If you look back you will see that the Data Direction register address is 2 less than its associated Control register address. This being the case, we load a mask into Accumulator A, then output it to the address which equals CX-2. Having loaded the necessary mask into the Data Direction

register, we restore the initial Control register contents. which we assumed contained 1 in bit position 2. Now another instruction that writes to address CX-2 will access an I/O port rather than a Data Direction register.

We have defined the purpose of Control register bit 2.

What about the remaining Control register bits?

You must write Control codes into these remaining bits in order to determine the manner in which each I/O port of an MC6820 will operate. You have these three options:

1) You may select simple data transfer without any handshaking control signals. in which case control signals are ignored.

2) You may select data input or data output with automatic handshaking protocol. The automatic handshaking protocol is dependent on control signals CA1 and CA2 for I/O Port A and control signals CB1 and CB2 for I/O Port B. As stated earlier, I/O Port A will only handle data input with handshaking while I/O Port B will only handle data output with handshaking. In each case there are two automatic handshaking options; one uses interrupt logic while the other does not.

3) Instead of using automatic handshaking protocol, you can design your own handshaking protocol. Your own handshaking protocol can generate interrupt requests based on high-to-low or low-to-high transitions of control signals. In addition you can modify the levels of control signals CA2 and CB2 under program control. You cannot modify the levels of control signals CA1 and CB1 since these are input only signals.

Given these options, let us examine how the individual bits of each Control register function.

The two high order bits of each Control register are read-only locations, which record the status of interrupt requests which may originate from either of two control lines associated with an I/O port:

```
7  6  5  4  3  2  1  0  ◄────── Bit No.
┌──┬──┬──┬──┬──┬──┬──┬──┐
│  │  │  │  │  │  │  │  │   Control Register A
└──┴──┴──┴──┴──┴──┴──┴──┘
      └───────────────► Status of interrupt requests originating at CA2 logic
         └────────────► Status of interrupt requests originating at CA1 logic
```

```
7  6  5  4  3  2  1  0  ◄────── Bit No.
┌──┬──┬──┬──┬──┬──┬──┬──┐
│  │  │  │  │  │  │  │  │   Control Register B
└──┴──┴──┴──┴──┴──┴──┴──┘
      └───────────────► Status of interrupt requests originating at CB2 logic
         └────────────► Status of interrupt requests originating at CB1 logic
```
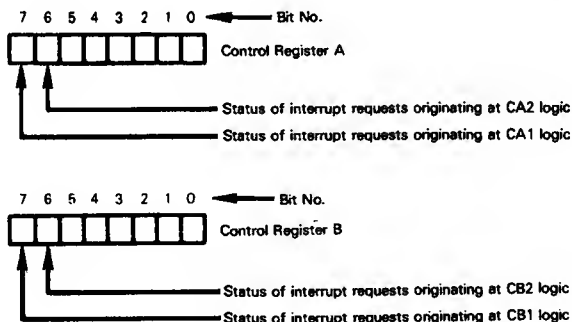
The remaining six control bits may be written into or read; they define the way in which the I/O port will operate.

Figures 3-3 and 3-4 describe the Control register interpretation for I/O Ports A and B respectively; since the two Control register interpretations are very similar. the points of difference are shaded so that they are easy to spot.

Let us clarify the functions enabled by the two Control registers.

If an I/O port is being used for simple data transfer without handshaking control signals, then load the following code into the I/O port Control register:

```
  7  6  5  4  3  2  1  0  ◄── Bit No.
┌──┬──┬──┬──┬──┬──┬──┬──┐
│X │X │0 │X │0 │1 │X │0 │  Control register for I/O Port A or B
└──┴──┴──┴──┴──┴──┴──┴──┘
                    │
              Disable interrupts
         Select I/O port rather than Data Direction register
      Don't care
```
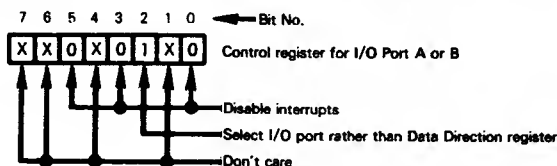
If you are going to use automatic interrupt or programmed handshaking, then you need to understand in more detail the interactions of the various Control register bits. These interactions are easy to understand if you bear in mind that the condition of bit 5 determines one of two possible interpretations for bits 4 and 3.

Let us consider the various handshaking options.

Each I/O port has its own interrupt request signal: $\overline{IRQA}$ for I/O Port A and $\overline{IRQB}$ for I/O Port B. Each interrupt request signal has two separate sets of request logic, based on an interrupt request originating with a CA1/CB1 signal transition, or a CA2/CB2 signal transition.

```
MC6820
INTERRUPT
LOGIC
```
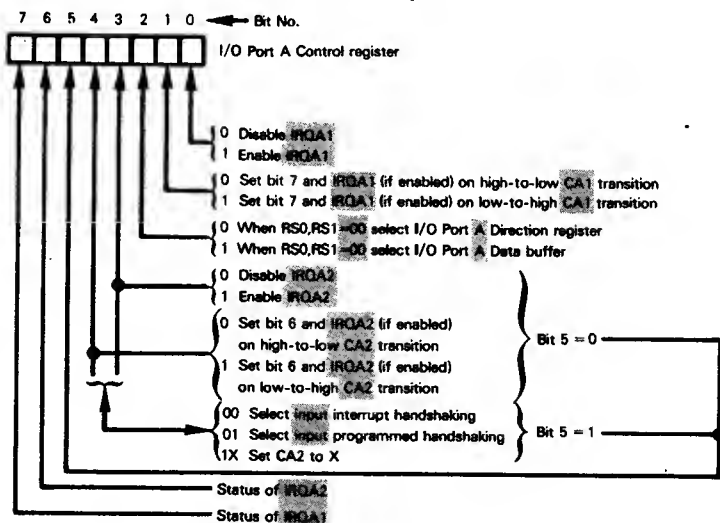

Figure 3-3. I/O Port A Control Register Interpretation

The figure shows:
- 0 Disable IRQA1
- 1 Enable IRQA1
- 0 Set bit 7 and IRQA1 (if enabled) on high-to-low CA1 transition
- 1 Set bit 7 and IRQA1 (if enabled) on low-to-high CA1 transition
- 0 When RS0,RS1=00 select I/O Port A Direction register
- 1 When RS0,RS1=00 select I/O Port A Data buffer
- 0 Disable IRQA2
- 1 Enable IRQA2
- 0 Set bit 6 and IRQA2 (if enabled) on high-to-low CA2 transition
- 1 Set bit 6 and IRQA2 (if enabled) on low-to-high CA2 transition } Bit 5 = 0
- 00 Select input interrupt handshaking
- 01 Select input programmed handshaking
- 1X Set CA2 to X } Bit 5 = 1
- Status of IRQA2
- Status of IRQA1

3-16

I/O Port B Control register

0 Disable IRQB1
1 Enable IRQB1

0 Set bit 7 and IRQB1 (if enabled) on high-to-low CB1 transition
1 Set bit 7 and IRQB1 (if enabled) on low-to-high CB1 transition

0 When RS0,RS1 = 01, select I/O Port B Direction register
1 When RS0,RS1 = 01 select I/O Port B Data buffer

0 Disable IRQB2
1 Enable IRQB2

0 Set bit 6 and IRQA2 (if enabled)
  on high-to-low CA2 transition   Bit 5 = 0
1 Set bit 6 and IRQA2 (if enabled)
  on low-to-high CA2 transition

00 · Select output interrupt handshaking
01 Select output programmed handshaking   Bit 5 = 1
1X Set CB2 to X

Status of IRQB2
Status of IRQB1

Figure 3-4. I/O Port B Control Register Interpretation

Control register bit 0 enables or disables $\overline{IRQA}/\overline{IRQB}$, based on signal CA1/CB1 transitions only. Quite independently, Control register bit 3 enables or disables $\overline{IRQA}/\overline{IRQB}$ based on transitions of signal CA2/CB2. However, Control register bit 3 has an alternative interpretation; the one we have just described only applies if Control register bit 5 is 0.

Interrupt requests are triggered by the "active transitions" of a control signal. The active transitions of control signals may be a high-to-low, or a low-to-high transition. For CA1/CB1, the active transition is selected by Control register bit 1. For CA2/CB2, the active transition is selected by Control register bit 4, but only if Control register bit 5 is 0.

Irrespective of whether interrupt request signals $\overline{IRQA}$ and $\overline{IRQB}$ have been enabled or disabled, Control register bits 6 and 7 will report the interrupt request as a status; that is to say, if a condition exists where CA1/CB1 makes an interrupt requesting active transition, then Control register bit 7 will be set to 1. Similarly, if control signal CA2/CB2 makes an interrupt requesting transition, then Control register bit 6 will be set to 1. Once set, Control register bits 6 and 7 will remain set until a Read operation addresses the Control register; at that time Control register bits 6 and 7 will both be reset to 0, while other bits of the Control register are left unaltered.

If Control register bit 5 is 1, then Control register bits 4 and 3 take on a second interpretation. If Control register bits 5 and 4 are both 1, then control signal CA2/CB2 will be output at all times with the level of Control register bit 3.

Consider the automatic handshaking options of the MC6820.

If Control register bits 5 and 4 are 1 and 0 respectively, then Control register bit 3 specifies an automatic handshaking signal sequence. Let us describe these signal sequences.

MC6820
AUTOMATIC
HANDSHAK-
ING

Input interrupt handshaking applies to I/O Port A only, and may be illustrated as follows:
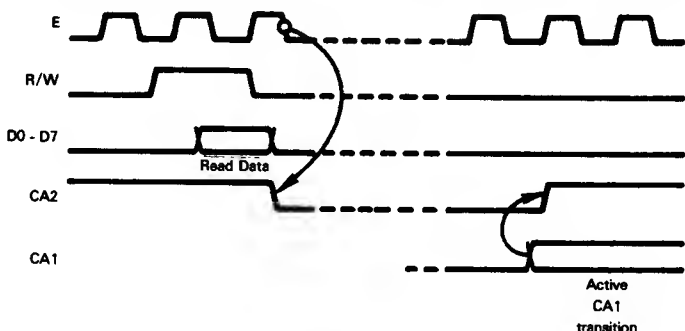
CA2 is output on the trailing edge of E, after the CPU has read the contents of the I/O Port A data buffer; this tells external logic that previously input data has been read and new data may now be input. External logic receives CA2 low, and upon transmitting new data to I/O Port A, must cause an active interrupt requesting transition of input control signal CA1. What constitutes an active transition will be determined by I/O Port A Control register bit 1. When external logic requests an interrupt via signal CA1, CA2 will be set high again.

Input programmed handshaking applies only to I/O Port A, and may be illustrated as follows:
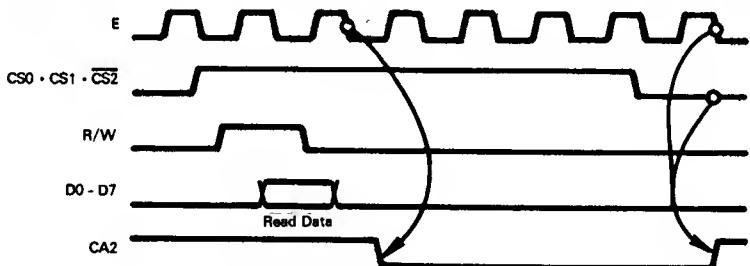
Once again control signal CA2 is output low when I/O Port A data buffer contents are read by the CPU. This tells external logic that previously input data has been read and new data may be input. External logic does not have to identify newly transmitted data with an interrupt request; rather, CA2 will be reset as soon as the MC6820 is deselected. Using programmed handshaking, external logic may use the CA2 low pulse as a Write strobe, causing new data to be input to I/O Port A.

Output interrupt handshaking applies only to I/O Port B, and may be illustrated as follows:

E

R/W

D0 - D7
Write Data

CB2

CB1
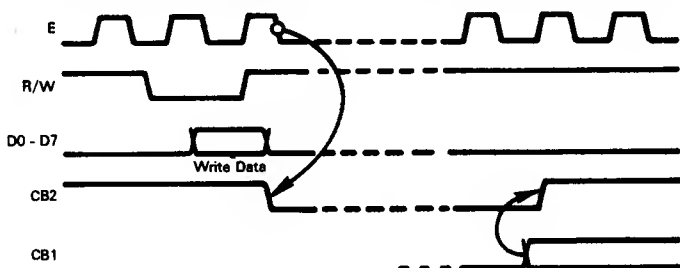
In this instance, control signal CB2 is output low on the high-to-low transition of E following a Write to I/O Port B Data buffer. In other words, CB2 tells external logic that new data has been output to I/O Port B and is ready to be read. External logic tells the MC6820 that I/O Port B contents have been read by making an interrupt requesting active transition of the CB1 signal. Once again, I/O Port B Control register bit 1 will determine what constitutes an active transition of the CB1 signal. Program logic can use an interrupt to branch to a program which outputs the next byte of data to I/O Port B.

Output programmed handshaking applies only to I/O Port B, and may be illustrated as follows:

E

CS0 · CS1 · $\overline{CS2}$

R/W

D0 - D7
Write Data

CA2
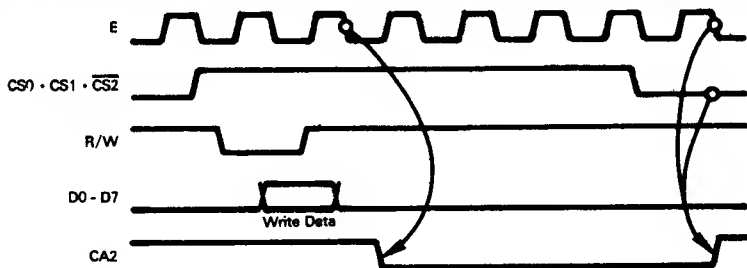
CB2 makes a high-to-low transition when data is written into the I/O Port B data buffer, just as occurred with output interrupt handshaking. However, CB2 will automatically be set to 1 as soon as the MC6820 is deselected. External logic can use the CB2 low pulse as a strobe, causing it to read the contents of I/O Port B.

Now consider how you can create your own handshaking options.

When you are creating your own handshaking control signals, bit 5 will be set to 0. Bit 0 will be set to 1 in order to enable interrupts generated by CA1 or CB1; bit 3 will be set to 1 in order to enable interrupts generated by CA2 or CB2. The levels of bits 1 and 4 determine whether a high-to-low transition or a low-to-high transition of control signals will generate an interrupt request.

You can at any time modify the level being output via CA2 or CB2 as follows:

XX110XXX   output to a Control register sets CA2 or CB2 low
XX111XXX   output to a Control register sets CA2 or CB2 high

X represents a "don't care" bit, but should be left equal to whatever value was previously in that bit position.

INPUT SIGNALS

Let us now assume that input and output signals are handled via four MC6820 PIA I/O ports, which we will identify as I/O Ports A, B, C and D, with these memory addresses:

$$
\begin{array}{lll}
\text{I/O Port} & \text{A} & \text{address is} & C000_{16} \\
& \text{B} & & C001_{16} \\
& \text{C} & & E000_{16} \\
& \text{D} & & E001_{16} \\
\end{array}
$$

Let us turn our attention to the input signals that appear on the left-hand side of Figure 3-1. We will describe each signal, assign it to an appropriate input pin, and include a rudimentary instruction sequence to access the signal at the most elementary level.

RETURN STROBE

If the operator is to see the most recently printed character, two things must happen:

1)   The printwheel must be moved to its position of visibility.

2)   The ribbon must be dropped.

External logic can take care of dropping and raising the ribbon, but logic in Figure 3-1 creates the signals that allow the printwheel to move.

In order to move the printwheel to its position of visibility, therefore, the ribbon control external logic inputs RETURN STROBE low while the ribbon is dropped.

Logic within Figure 3-1 uses RETURN STROBE as an alternative signal to start a print cycle; however, RETURN STROBE low is accompanied by HAMMER ENABLE FF low, which prevents the printhammer from firing. Therefore **a print cycle initiated by RETURN STROBE low is a**

| PRINTWHEEL REPOSITION- ING PRINT CYCLE |

"dummy" print cycle which moves the printwheel back to its position of visibility, but does not fire the printhammer; we refer to this as a printwheel repositioning print cycle:
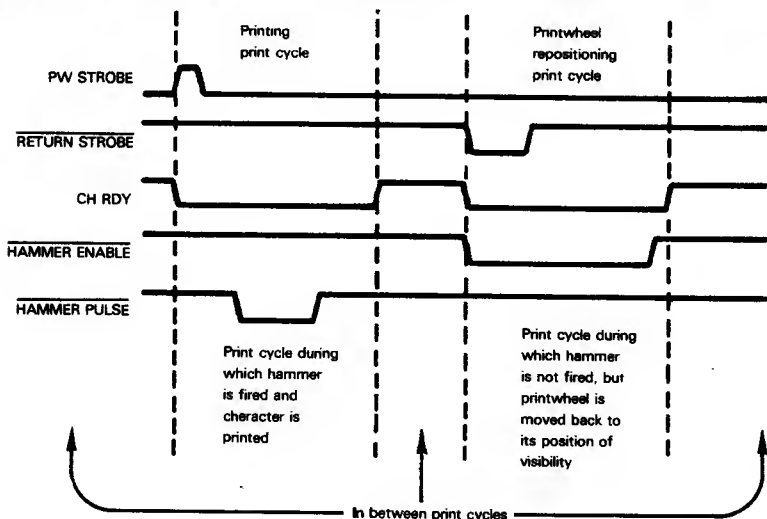
We will assign I/O Port B, pin 4 to RETURN STROBE.

In between print cycles, we can test this pin in order to trigger a new print cycle via the following instruction sequence:

```
LOOP    LDA A    $C001        INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$10         MASK OUT ALL BAR BIT 4
        BNE      LOOP         IF THIS BIT IS 1, RETURN AND RETEST
;NEW PRINT CYCLE INSTRUCTION SEQUENCE BEGINS HERE
```

Here is a check on how the AND instruction works in the sequence above:

```
        7 6 5 4 3 2 1 0 ◄── Bit No.
        X X X Y X X X X     Accumulator A contents
        0 0 0 1 0 0 0 0     #$10
        ─────────────
        0 0 0 Y 0 0 0 0     AND

              ▲───────────── This bit corresponds to RETURN STROBE
```

PFL REL

The printhammer cannot be fired while the paper feed mechanism is moving, therefore at such times external logic inputs PFL REL low.

Logic within Figure 3-1 will delay firing the printhammer for as long as PFL REL is being input low.

We will assign Pin 0 of I/O Port A to PFL REL.

Before executing the instruction sequence which fires the printhammer, we will input the contents of Port A and test bit 0; so long as this bit contains zero, we will not execute the printhammer firing sequence.

The following instructions perform the required test:

```
LOOP    LDA A    $C000        INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        AND A    #1           MASK OUT ALL BITS BAR BIT 0
        BEQ      LOOP         THE BIT IS 0, DO NOT FIRE THE PRINTHAMMER
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

RIB LIFT RDY

This signal is similar to PFL REL; it **is input low when ribbon lift logic is moving the ribbon.** Just as the printhammer cannot be fired while the paper feed mechanism is active, so it cannot be fired while the ribbon is being moved. By connecting RIB LIFT RDY to Pin 1 of I/O Port A, we may adjust the printhammer firing initiation instruction sequence as follows:

```
LOOP    LDA A    $C000        INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        ORA A    #$FC         MASK OUT ALL BITS BAR 0 AND 1
        COM A                 COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
                              PRESENT
        BNE      LOOP         ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS NOW 1, DO
                              NOT FIRE PRINTHAMMER
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

PW STROBE

We have already encountered **this signal; it is pulsed high by external logic to start a normal print cycle,** during which a character will be printed.

Remember, RETURN STROBE is input low to initiate a print cycle during which the printwheel will be moved to its position of visibility, but no character will be printed.

Assuming that **PW STROBE is connected to pin 5 of I/O Port B,** this is the instruction sequence that will be executed between print cycles:

```
LOOP    LDA A    $C001    INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$30     ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CMP A    #$10     TEST FOR PW STROBE=0. RETURN STROBE=1
        BEQ      LOOP     IF TEST IS TRUE, STAY IN LOOP
PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

Observe that either PW STROBE = 1, or $\overline{\text{RETURN STROBE}}$ = 0 can trigger the start of a print cycle; that is why only PW STROBE = 0 and $\overline{\text{RETURN STROBE}}$ = 1 keeps us in the testing instruction loop.

Now the four instructions shown above execute in a combined total of 12 clock cycles. With a one microsecond clock, the four instructions will execute in 12 microseconds — which becomes the minimum pulse width allowed for PW STROBE. **If PW STROBE is pulsed high for less than 12 microseconds, our instruction cycle may miss it.**

> **INPUT SIGNAL PULSE WIDTH**

$\overline{\text{FFA}}$

This is another printhammer warning **signal. It is set to 0 while external logic is advancing the ribbon. By connecting this signal to pin 2 of I/O Port A, we can modify the instruction sequence which precedes printhammer firing as follows:**

```
LOOP    LDA A    $C000    INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        ORA A    #$F8     ISOLATE BITS 2, 1 AND 0
        COM A    .        COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        BNE      LOOP     ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                          PRINTHAMMER
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

All we have done is add one more test condition which must be met before the printhammer firing instruction sequence gets executed.

RESET

This is a signal which is commonly seen in the most diverse types of logic. It is **an initializing signal.** Its purpose is to ensure that all logic is in a "beginning" state, which in our case is the condition which exists between printwheel cycles.

The logic in Figure 3-1 connects the RESET signal to logic devices such that RESET going high forces all logic to a "beginning" condition.

There are many ways in which a microcomputer system can handle a RESET signal. The simplest scheme is to input this signal to the $\overline{\text{RESET}}$ pin of the MC6800 CPU.

> **RESET THE CPU**

Another method of handling RESET is to test the signal in between print cycles and to prevent any print cycle from starting while RESET is high; this may be accomplished by connecting RESET to pin 6 of I/O Port B, then modifying our "in between print cycles" instruction sequence as follows:

```
LOOP    LDA A    $C001    INPUT I/O PORT B CONTENTS TO ACCUMULATOR A  .
        AND A    #$40     ISOLATE BIT 6 (RESET)
        BNE      LOOP     IF RESET IS HIGH, STAY IN LOOP
RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
        LDA A    $C001    INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$30     ISOLATE BIT 5 (PW STROBE) AND BIT 4 (RETURN STROBE)
        CMP A    #$10     TEST FOR PW STROBE=0, RETURN STROBE=1
        BEQ      LOOP     IF TEST IS TRUE STAY IN LOOP
PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

Now this longer test loop will require 22 cycles to execute. That means PW STROBE must pulse high for at least 22 microseconds, assuming a 1 microsecond clock.

PFR REL

This is yet another **signal** which must be tested before initiating printhammer firing. It **indicates when external logic is moving the paper feed.** Under such circumstances we cannot fire the printhammer. **By connecting this signal to pin 3 of input Port A, we merely have to adjust the printhammer firing instruction initiation sequence as follows:**

```
LOOP    LDA A    $C000      INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        ORA A    #$F0       ISOLATE BITS 3, 2, 1 AND 0
        COM A               COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        BNE      LOOP       ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                            PRINTHAMMER
```
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

CA REL

This **signal** is almost identical to PFR REL. It **comes from external logic that controls carriage movement. We will connect this signal to pin 4 of input Port A** and modify the hammer firing instruction initiation sequence as follows:

```
LOOP    LDA A    $C000      INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        ORA A    #$E0       ISOLATE BITS 4, 3, 2, 1 AND 0
        COM A               COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        BNE      LOOP       ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                            PRINTHAMMER
```
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

FFI

This is the signal which times the first delay in the print cycle — the time during which the printwheel moves from its position of visibility until the required petal is in front of the printhammer.

FFI is generated by external logic; it is low while the printwheel is moving and it is high while the printwheel is not moving.

We will tie FFI to pin 7 of I/O Port A. The following instruction loop will create a delay which lasts until FFI goes high:

```
LOOP    LDA A    $C000      INPUT PORT A TO ACCUMULATOR A
        ROL A               SHIFT BIT 7 INTO THE CARRY
        BCC      LOOP       IF CARRY = 0 STAY IN THE LOOP
```

Do you see how this loop works? After I/O Port A contents have been input to Accumulator A, we are only interested in bit 7, since this is the bit that corresponds to FFI.

This is what the ROL A instruction does:
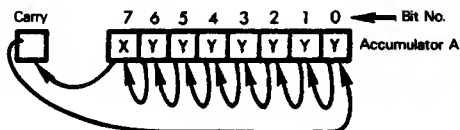
If the Carry status equals 1, the printwheel move delay is over. If Carry equals 0, program logic must continue the delay.

EOR DET

This signal indicates that the end of the ribbon has been reached. Under these circumstances character printing cannot continue.

When this signal is generated, there will still be fresh ribbon in front of the printhammer, so the signal is not used to inhibit printhammer firing; rather it is used to prevent the end of the print cycle from ever being indicated. This effectively prevents a new print cycle from ever starting.

We will connect the EOR DET signal to bit 7 of I/O Port B. Since EOR DET is a negative logic signal we will test it prior to going into the "in between print cycle" loop as follows:

```
TEST FOR VALID END OF PRINT CYCLE
LOP1    LDA A    $C001      INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        ROL A               SHIFT BIT 7 INTO CARRY
        BCC      LOP1       IF ZERO IN CARRY, STAY IN PRINT CYCLE
START OF IN BETWEEN PRINT CYCLES LOOP
LOOP    LDA A    $C001      INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$40       ISOLATE BIT 6 (RESET)
        BNE      LOOP       IF RESET IS HIGH, STAY IN LOOP
RESET IS LOW SO TEST PW STROBE AND RETURN STROBE
        LDA A    $C001      INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$30       ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CMP A    #$10       TEST FOR PW STROBE=0, RETURN STROBE=1
        BEQ      LOOP       IF TEST IS TRUE STAY IN LOOP
PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

Look at the above instruction sequence. There are some interesting aspects to it.

The first three instructions above will be the last three instructions in the print cycle sequence. The instruction labeled LOOP is the first instruction of a sequence which gets executed continuously until the start of the next print cycle. Thus, if EOR DET is low, program logic will hang up in the first three instructions listed above, constantly looping within these three instructions until EOR DET goes high. At that time, the print cycle ends and we go into the "in between print cycles" instruction loop. The program now hangs up indefinitely in this instruction loop until bit 6 which corresponds to RESET equals 0, while bit 5 which corresponds to PW STROBE equals 1, or bit 4 which corresponds to RETURN STROBE equals 0.

There is another interesting feature of the above instruction sequence. We could, if we wished, eliminate the second LDA instruction, as follows:

```
TEST FOR VALID END OF PRINT CYCLE
LOP1    LDA A    $C001      INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        ROL A               SHIFT BIT 7 INTO CARRY
        BCC      LOP1       IF ZERO IN CARRY, STAY IN PRINT CYCLE
START OF IN BETWEEN PRINT CYCLES LOOP
        AND A    #$80       ISOLATE BIT 6 (RESET)
        BNE      LOP1       IF RESET IS HIGH, STAY IN LOOP
RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
        LDA A    $C001      INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A    #$30       ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
        CMP A    #$10       TEST FOR PW STROBE=0, RETURN STROBE=1
        BEQ      LOP1       IF TEST IS TRUE STAY IN LOOP
PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

By eliminating one instruction, we have saved two bytes of object code. The penalty is that we have added six clock cycles to the entire instruction loop, which means that the PW STROBE high pulse goes up from the 22 microseconds we calculated when discussing the RESET signal to 28 microseconds.

Why does the condensed instruction sequence illustrated above work? The reason is because external logic is not supposed to be moving the ribbon in between print cycles, therefore EOR DET will always be high during the in between print cycle instruction execution loop. If this is so, the ROL instruction will always shift a 1 into the Carry, which will always cause execution to continue with the AND instruction. Thus the first three instructions become harmless. Notice that the AND A #$40 instruction has become an AND A #$80 instruction since the RESET signal bit has been shifted one position to the left by the ROL instruction.

HAMMER ENABLE FF.

This is the signal which prevents the printhammer from being fired after the printwheel is moved to its position of visibility, as described in connection with the RETURN STROBE signal.

We will connect HAMMER ENABLE FF to pin 6 of I/O Port A, then modify the instruction sequence which precedes printhammer firing as follows:

```
LOOP    LDA A   $C000       INPUT CONTENTS OF I/O PORT A TO ACCUMULATOR A
        ORA A   #$A0        ISOLATE BITS 6, 4, 3, 2, 1 AND 0
        COM A               COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        BNE     LOOP        ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO
                            NOT FIRE PRINTHAMMER
PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

CLK

This is the clock signal that synchronizes all logic in Figure 3-1. Try as we may, we cannot include this signal in our simulation of Figure 3-1, since events within the microcomputer program are going to be synchronized by the sequence in which instructions are executed — not by a clock. **Similarly, the next two signals, +5V and RV1, are power supplies. They are meaningless within a microcomputer program.**

H1 - H6

These are the six signals which select one of six time durations for the printhammer firing pulse. We will assign these signals to I/O Port C. Once the printhammer firing instruction sequence gets executed, it simply loads these signals into Accumulator B as follows:

```
        LDA B   $E000       INPUT FIRING PULSE TIME CODE TO ACCUMULATOR B
```

INPUT SIGNAL SUMMARY

In summary, this is how input signals have baen assignad:

MC6820 Port A assigned to input		
	7	FFI
	6	HAMMER ENABLE
	5	
	4	CA REL
	3	PFR REL
	2	FFA
	1	RIB LIFT RDY
	0	PFL REL

MC6820 Port B assigned to input		
	7	EOR DET
	6	RESET
	5	PW STROBE
	4	RETURN STROBE

MC6820 Port C assigned to input		
	7	
	6	
	5	H6
	4	H5
	3	H4
	2	H3
	1	H2
	0	H1

OUTPUT SIGNALS

Wa will now turn our attention to the output signals listed on the right-hand side of Figure 3-1. These signals are much easier to describe than the input signals. They consist of six flip-flop outputs — which are simply timing indicators used by external logic — plus four control signals. **We are going to output these signals to the B port of ona MC6820 PIA and tha D port of the second MC6820 PIA as follows:**

MC6820 Port D assigned to output		
	7	
	6	FFF
	5	FFE
	4	FFE
	3	FFD
	2	FFC
	1	FFB
	0	FFA

MC6820 Port B assigned to output		
	3	START RIB MOTION
	2	HAMMER PULSE
	1	CH RDY
	0	PW RELEASE

We assign a pin for FFC even though it is not output, because I/O Port D is going to serve a double purpose — as a data storage location and as an output signals' buffer. Simple routines to generate output signals cannot be concocted; that is the whole purpose of the logic in Figure 3-1. We will therefore simply define the four output control signals:

1) **PW REL. This signal marks the and of the fixed printhammer return and settling time delay and the beginning of the fixed Final Movement's delay** during which external logic can move the paper feed and carriage.

2) **CH RDY. This is also referred to as the PRINTWHEEL READY signal. This is the signal which defines the entire print cycle time interval;** it goes low at the start of the print cycle and stays low until the end of the print cycle.

3) **HAMMER PULSE. This signal must be output low for the time interval during which external logic is supposed to transmit a firing pulse to the printhammer solenoid.**

4) **START RIBBON MOTION PULSE. This signal is pulsed high early in the print cycle, telling external logic that it is safe to begin advancing the ribbon so that fresh ribbon will be in front of the printhammer when it is fired.**

A DIGITAL LOGIC ORIENTED SIMULATION

We are now ready to start simulating the logic illustrated in Figure 3-1 — but first a brief overview of the logic.

A LOGIC OVERVIEW

At the center of the logic sequence are four 74107 flip-flops, labeled FFC$_W$, FFD$_W$, FFE$_W$ and FFF$_W$. You will find these flip-flops in the center and to the left of Figure 3-1. **These four flip-flops form what is known as a "Johnson Counter".** Each flip-flop is controlled by the output of the previous flip-flop, coupled with a test for external conditions:
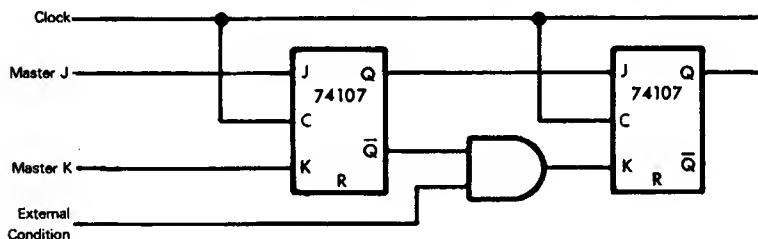
Thus the four flip-flops may be visualized as initiating print cycle events in the following way:
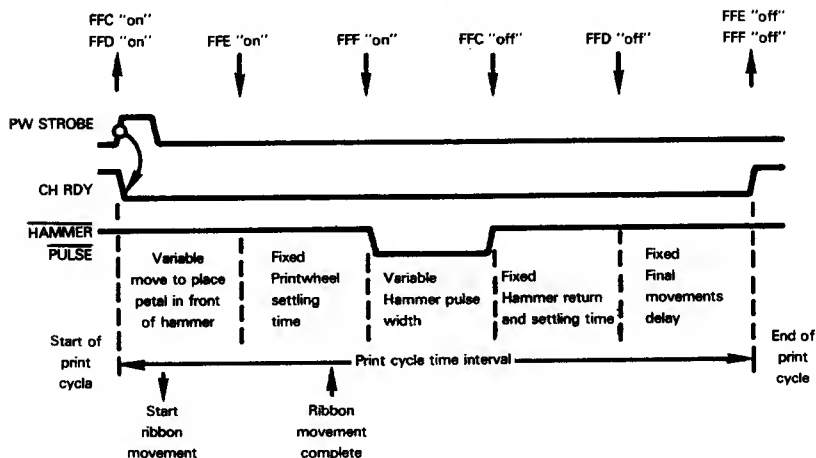
As illustrated above, the print cycle time interval may be divided into five periods.

During the first time interval the printwheel is moved from its position of visibility until the required petal is in front of the printhammer. This time interval is controlled by external logic, via the FFI input.

The remaining four time intervals are controlled by three 74121 one-shots and the 555 multivibrator.

What about the two 7474 flip-flops at the top left-hand corner of Figure 3-1? These are simply cycle initiation logic. Flip-flop FFA is triggered by a combination of signals necessary for a print cycle to begin. Flip-flop FFB acts as a switch for the four 74107 flip-flops, forcing them to turn "off" in between print cycles. Flip-flop FFB does this by tying its Q output to the reset inputs of the 74107 flip-flops. This results in the 74107 flip-flops always being turned off if FFB is turned off; we will explain in more detail how this happens later on.

We are now going to follow a print cycle through Figure 3-1. As we progress, we will create a microcomputer assembly language program that simulates the logic, device-by-device.

FLIP-FLOP FFA$_W$

Our print cycle begins at the 7474 flip-flop designated FFA$_W$. You will find this flip-flop at the top, left-hand corner of Figure 3-1. Let us isolate FFA$_W$, and illustrate it as follows:

| 7474 |
| FLIP-FLOP |

((CH RDY) OR (PW STROBE)) AND (NOT RESET)
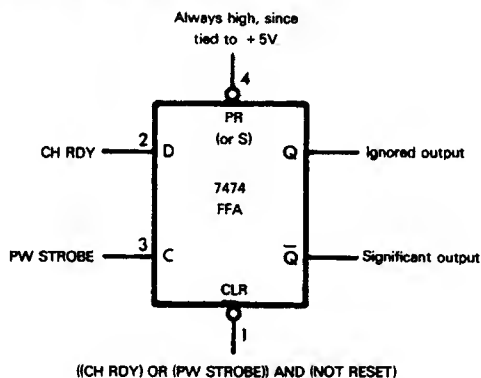
Refer back to the general function table for a 7474 flip-flop, given in Chapter 2.

Since PRESET (PR) is always high, being tied to $+5V$, a low CLEAR (CLR) input will force the flip-flop "off", at which time Q is output low and $\overline{Q}$ is output high.

Look at Figure 3-1 and you will see that CLR is generated as follows:
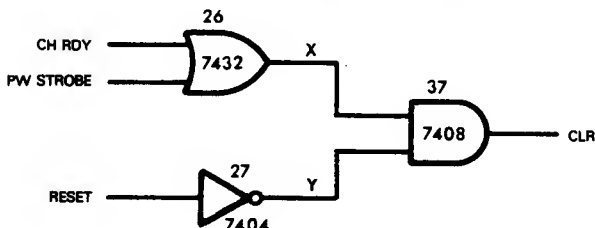
This is the truth table for CLR:

CH RDY	PW STROBE	X	RESET	Y	CLR
0	0	0	0	1	0
			1	0	0
0	1	1	0	1	1
			1	0	0
1	0	1	0	1	1
			1	0	0
1	1	1	0	1	1
			1	0	0

For flip-flop FFA$_W$ to turn "on", CLR must be high; for CLR to be high, RESET must be low, and either CH RDY or PW STROBE must be high.

Now CH RDY provides FFA$_W$ with its data (D) input, and PW STROBE provides the clock (C) input. Therefore the function table for flip-flop FFA$_W$ may be illustrated as follows:

INPUTS				OUTPUTS		
PRESET	CLR	CLOCK (PW STROBE)	D (CH RDY)	Q	Q̄	
0	1	0 or 1	0 or 1	1	0	PRESET=1
1	0	0 or 1	0 or 1	0	1	
0	0	0 or 1	0 or 1	Unstable		PRESET=1
1	1	0 → 1	1	1	0	
1	↑	0 → 1	0	0	1	
1	1	0	0 or 1	Previous Q	Previous Q̄	No change

And this devolves to the following small function table:

CLR	CH RDY	PW STROBE	Q̄	
0			1	"off" condition
1	0	0 → 1	1	possible "on" conditions
1	1	0 → 1	0	

It takes a 0 to 1 transition of PW STROBE for flip-flop FFA$_W$ to turn on. When FFA$_W$ turns on, however, if CH RDY is 0, then the Q̄ output is still 1, representing the "off" condition. Thus, to turn FFA$_W$ "on", PW STROBE must go from 0 to 1 while CH RDY is 1.

Recall that CH RDY is a signal which is output high in between print cycles and is output low for the duration of a print cycle. This means that flip-flop FFA$_W$ will only turn on if PW STROBE pulses high in between print cycles, as characterized by CH RDY being output high:

CLR	CH RDY	PW STROBE	Q̄
0			1
1	0	0 → 1	1
1	1	0 → 1	0

PW STROBE

CH RDY

In between Print Cycles

Print Cycles

Start of Print Cycle
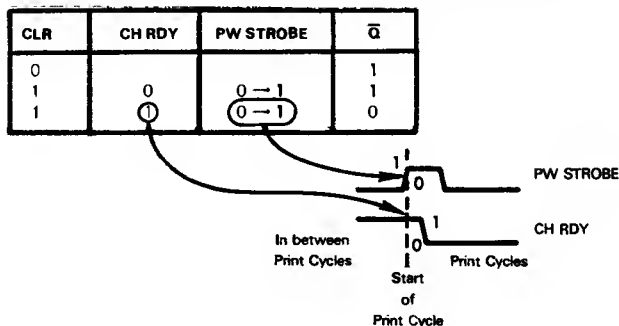
For the moment do not worry about how CH RDY goes to 0 shortly after flip-flop FFA$_W$ turns on. We will explain how this happens later. The only important thing to note is that a PW STROBE high pulse will be ignored if it occurs while CH RDY is low.

What about the RESET signal? What this signal does is over- | RESET |
ride all other logic associated with flip-flop FFA$_W$; whenever
RESET is input high, CLR is forced low which turns flip-flop FFA$_W$ off irrespective
of whatever else is going on.

SIMULATING FLIP-FLOP FFA$_W$

We concluded in Chapter 2 that a flip-flop is represented in a microcomputer system by a single bit of read/write memory. A single bit of a read/write buffer will do just as well.

I/O Port D has been assigned to output signals. This port has an | **FLIP-FLOP**
8-bit buffer to which port pins are connected; thus **each bit of the port** | **SIMULATION**
buffer will simulate the flip-flop whose output is transmitted | **USING I/O**
via the port pin: | **PORTS**
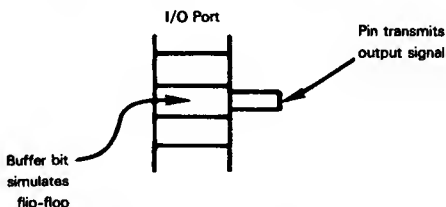
Recall that FFA has been assigned pin 0 of I/O Port D.

O.K., we are ready to simulate flip-flop FFA$_W$.

At the same time, how about simulating the three gates below and to the left of FFA$_W$? These three gates are numbered 26, 27 and 37 and together they create the CLR input.

Simulating these three gates individually, the following instruction sequence applies:

```
SIMULATE GATE 27
        LDA  B    $C001        INPUT I/O PORT B CONTENTS TO ACCUMULATOR B
        COM B                  COMPLEMENT ALL EIGHT BITS
        AND B     #$40         ISOLATE BIT 6; IT REPRESENTS RESET COMPLEMENT
SIMULATE GATE 26
        LDA  A    $C001        INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        AND A     #$22         ISOLATE BITS 5 AND 1; THEY REPRESENT PW
                               STROBE AND CH RDY
SIMULATE GATE 37
        BEO       CLR0         IF NEITHER BIT 1 NOR 5 EQUALS 1, CLR IS 0
        BIT  B    #$FF         TEST COMPLEMENT OF RESET
        BEO       CLR0         IF RESULT IS 0, CLR IS 0
        SEC                    CLR IS 1 SO STORE 1 IN CARRY STATUS
        JMP       FFAW + 2
CLR0    CLC                    CLR is 0 SO STORE 0 IN CARRY STATUS
SIMULATE FLIP-FLOP FFAW
```

```
FFAW    BCC     FFA0        IF CLR IS 0, SET I/O PORT D, BIT 0 TO 1
        BIT  A  #$20        CLR IS NOT 0. TEST PW STROBE. IF PW STROBE
                            IS 0, CLOCK HAS NOT PULSED
        BEQ     FFA0        SET BIT 0 OF I/O PORT D TO 1
        BIT  A  #$02        PW STROBE IS 1, TEST CH RDY
        BEQ     FFA0        IF CH RDY IS 0, SET BIT 0 OF I/O PORT D TO 1
        LDA  A  $E001       LOAD I/O PORT D INTO ACCUMULATOR A
        AND  A  #$FE        BIT 0 MUST BE RESET TO 0, SINCE FFA IS "ON"
        STA  A  $E001
        JMP     FFB         JUMP TO FLIP-FLOP B SIMULATION
FFA0    LDA  A  $E001       LOAD I/O PORT D INTO ACCUMULATOR A
        ORA  A  #1          BIT 0 MUST BE SET TO 1 SINCE FFA IS "OFF"
        STA  A  $E001
FLIP-FLOP FFB SIMULATION FOLLOWS
```

It is very important that you understand how instructions fit together to make a program. Read no further until you understand completely how the instruction sequence given above simulates the logic of FFA$_W$ and its three associated gates.

Let us look at the above simulations.

The RESET signal, you will recall, has been tied to bit 6 of MC6B20 I/O Port B; this port is addressed as memory location C001$_{16}$ based on the way in which we have elected to wire the MC6B20 PIA into our

> **INVERTER**
> **SIMULATION**

microcomputer system. In order to invert this signal, we input the contents of I/O Port B to Accumulator B, complement the contents of the Accumulator, then isolate the complement of RESET by setting all bits of Accumulator B to 0, bar bit 6.

```
                        from I/O Port B
        LDA  B  $C001       X X X X X X X X   to Accumulator B
        COM  B              X̄ X̄ X̄ X̄ X̄ X̄ X̄ X̄   Complement
        AND  B  #$40        0 1 0 0 0 0 0 0   Isolate bit 6
                            0 X̄ 0 0 0 0 0 0
```

The complement of RESET is saved in Accumulator B. **The simulation of gate 27 is complete.**

The simulation of gate 26 is not quite as straightforward.
We are seeking the OR of PW STROBE and CH RDY. These two signals are represented by bits 5 and 1, respectively, of I/O Port B. Now what we do is load the contents of I/O Port B into Accumulator A, then execute an AND instruction which sets all bits to 0, bar bits 5 and 1. But we do not actually OR these two remaining bits. Why? The

> **OR GATE**
> **SIMULATION**
>
> **STATUS FLAGS**
> **USED TO**
> **REPRESENT**
> **LOGIC**

reason is because when the AND instruction is executed, it sets the Zero status to the complement of (PW STROBE) OR (CH RDY):

A5 OR A1	A7	A6	A5	A4	A3	A2	A1	A0	HEX VALUE	ZERO STATUS
0	0	0	0	0	0	0	0	0	00	1
1	0	0	0	0	0	0	1	0	02	0
1	0	0	1	0	0	0	0	0	20	0
1	0	0	1	0	0	0	1	0	22	0

Accumulator A Contents

PW STROBE   CH RDY   Following AND instruction execution, Zero status is complement of (PW STROBE) OR (CH RDY).

We can therefore move on to gate 37.

The purpose of gate 37 is to generate the FFA$_W$CLR input. We are going to simulate CLR using the Carry status. Now we come right out of the gate 26 simulation into the gate 37 simulation; at this time the Zero status

ZERO STATUS

will be 0 if the OR of PW STROBE and CH RDY is 1; Zero status will be 1 otherwise. (Recall that Zero statuses always represent the inverse of the 0 condition. In other words, a 0 condition causes the Zero status to be set to 1; a nonzero condition causes the Zero status to be set to 0.)

The first instruction of the gate 37 simulation takes advantage of the fact that we have the OR of PW STROBE and CH RDY recorded in the Zero status. If the Zero status is 1, CLR must be 0, so the first BEQ instruction branches to logic that will set the Carry status to 0. The next instruction in the gate 37 simulation tests the complement of RESET as stored in Accumulator B, using a BIT instruction. The BIT instruction will not change the contents of Accumulator B, but it will reset statuses based on the result of an AND. If the complement of RESET is 0, then the BEQ instruction which follows will branch to program logic which sets the Carry status to 0. If the complement of RESET is not 0, then all conditions have been met for gate 37 to output a nonzero result — and this condition is simulated by the SEC instruction, which sets the Carry status to 1.

Flip-flop FFA is simulated next. The state of this flip-flop may be defined as follows:

  If CLR is 0 then $\overline{Q}$ is 1.
  If PW STROBE is 0 then $\overline{Q}$ is 1.
  If CLR is 1 and PW STROBE is 1 and CH RDY is 0 then $\overline{Q}$ is 1.
  If CLR is 1 and PW STROBE is 1 and CH RDY is 1 then $\overline{Q}$ is 0.

CLR is simulated by the Carry status. PW STROBE is simulated by bit 5 of Accumulator A. CH RDY is simulated by bit 1 of Accumulator A.

The simulation of flip-flop FFA begins with the instruction labeled FFAW.

First we test the status of CLR using the BCC instruction. This instruction causes a jump to FFA0 if the Carry status is 0 — which means that CLR is 0. FFA0 is the label for the first instruction in the sequence which sets $\overline{Q}$ to 1.

CARRY STATUS

Observe that we have some unnecessary steps at this point in the program. Here is our logic:

LDA	A	$C001
AND	A	#$22
BEQ		CLR0
BIT	B	#$FF
BEQ		CLR0
SEC		
JMP		FFAW + 2
CLR0	CLC	
FFAW	BCC	FFA0
BIT	A	#$20

```
              ┌─────────────────────┐
              │ Input I/O Port B     │
              │ to Accumulator A     │
              └─────────────────────┘
                        │
              ┌─────────────────────┐
              │ Isolate bits 5 and 1 │
              └─────────────────────┘
                        │
              ◇ If both are 0, branch ◇──── Both are 0 ──┐
              ◇ to CLR0               ◇                  │
                  │ Both are not 0                        │
              ┌─────────────────────┐                    │
              │ Test RESET in B      │                    │
              └─────────────────────┘                    │
                        │                                 │
              ◇ If RESET is 0, branch ◇──── RESET = 0 ───┤
              ◇ to CLR0               ◇                   │
                  │ RESET = 1                             │
              ┌─────────────────────┐                    │
              │ Set Carry to 1       │                    │
              │ signifying CLR = 1   │                    │
              └─────────────────────┘                    │
                        │                                 │
              ┌─────────────────────┐                    │
              │ Jump to PW STROBE    │                    │
              │ test                 │                    │
              └─────────────────────┘                    │
                                                          │
      (CLR0)  ┌─────────────────────┐                    │
              │ Set Carry to 0       │◄───────────────────┘
              │ signifying CLR = 0   │
              └─────────────────────┘
                        │
      (FFAW)  ◇ If Carry is 0, branch ◇──── Carry is 0 ──┐
              ◇ to FFA0               ◇                   │
                  │ Carry is 1                            │
              ┌─────────────────────┐                    │
              │ Carry is 1. Test     │                    │
              │ PW STROBE            │                    │
              └─────────────────────┘                    │
                        │                                 │
```

3-33
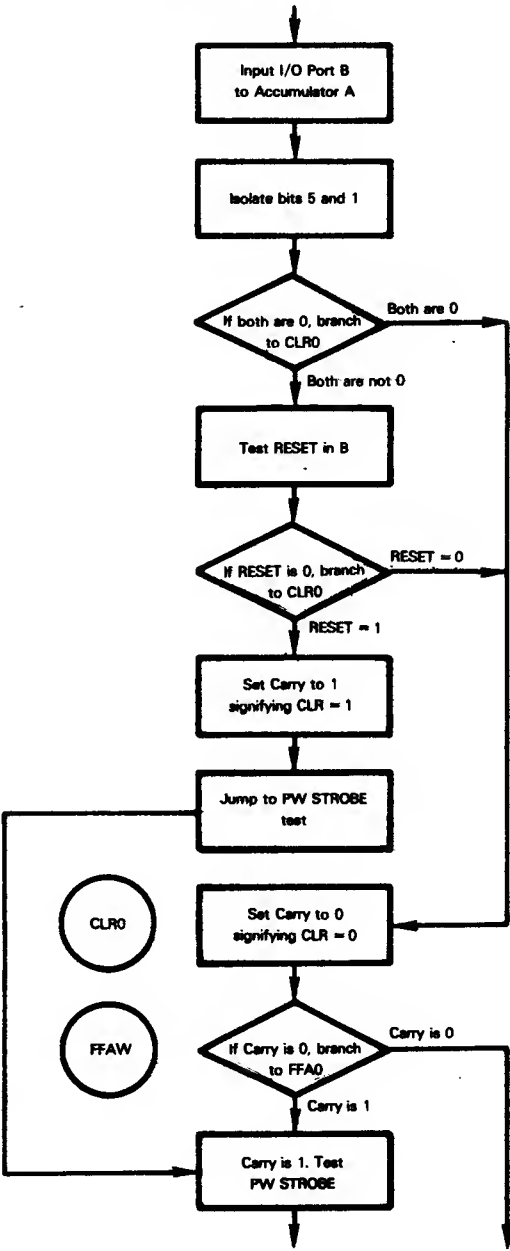
Each rectangular box represents a data movement or manipulation operation.

Each diamond represents logic which tests the condition of a status flag.

The logic sequence illustrated above maintains an orderly instruction flow which conforms with the flip-flop FFAW and its three preceding gates. But if you look at the instructions labeled CLR0 and FFAW, you will see that they are redundant. The instruction labeled CLR0 sets the Carry status to 0. The instruction labeled FFAW tests the Carry status, and upon detecting 0 branches to the later instruction labeled FFA0. But since we have just set the Carry status to 0, the instruction labeled FFAW must detect a 0 Carry status; therefore, the only allowed logic path following a branch to CLR0 is another branch to FFA0. We can therefore replace the two instructions which branch to CLR0 with instructions that branch directly to FFA0; then we can eliminate instructions labeled CLR0 and FFAW. This also eliminates the instruction which jumps to FFAW + 1, since FFAW + 1 addresses a BIT instruction which becomes the next sequential instruction. We can also remove the SEC instruction. Since Carry=0 conditions have been accounted for by branches to FFA0, the default is Carry=1, which no longer needs to be identified. Thus our new instruction sequence may be illustrated as follows:

Old Sequence			New Sequence		
	-			-	
	-			-	
	-			-	
LDA	A	$C001	LDA	A	$C001
AND	A	#$22	AND	B	#$22
BEQ		CLR0	BEQ		FFA0
BIT	B	#$FF	BIT	B	#$FF
BEQ		CLR0	BEQ		FFA0
SEC					
JMP		FFAW + 1			
CLR0	CLC		Unnecessary instructions		
FFAW	BCC	FFA0			
BIT	A	#$20	BIT	A	#$20
	-			-	
	-			-	
	-			-	

Let us continue our program analysis with the BIT A #$20 instruction.

Presuming that CLR has a value of 1, we next test PW-STROBE. Again, we use a BIT instruction for this purpose. PW STROBE is represented by bit 5 of Accumulator A. In order to test the status of this bit, the BIT instruction ANDs Accumulator A contents with a mask that contains a 1 in bit 5 and 0 in all other bit positions. The result of the AND is discarded — which means the contents of Accumulator A remain the same; however status bits are set or reset to reflect the result of the AND:

```
7 6 5 4 3 2 1 0 ◄────Bit No.
X X X X X X X X   Accumulator A contents
0 0 1 0 0 0 0 0   BIT mask
0 0 X 0 0 0 0 0   Result is discarded but is used to set status flags
     ▲
     └─────────► This bit represents PW STROBE
```

Assuming that PW STROBE is 1, all that remains is to check the condition of CH RDY. To do this we again execute a BIT instruction; however this time the contents of Accumulator A are ANDed with a mask that contains a 1 in bit 1 and 0 in all other bit positions. Again the result of the AND is discarded, which means that Accumulator A contents are not disturbed; however status flags are set or reset to reflect the result of the AND operation.

Assuming that all conditions have been met to turn flip-flop FFA on, we must set bit 0 of I/O Port D to 0. This is done by inputting the contents of I/O Port D to the Accumulator, ANDing with the appropriate mask, then returning the result:

```
7 6 5 4 3 2 1 0   ◄──── Bit No.
X X X X X X X Y         Accumulator A contents
1 1 1 1 1 1 1 0         #$FE
─────────────
X X X X X X X 0         AND
```

The last three instructions of the flip-flop FFA simulation are the three instructions which set bit 0 of I/O Port D to 1 (reflecting the fact that flip-flop FFA is "off"). These three instructions load the contents of I/O Port D into Accumulator A, OR with the appropriate mask, then return the result:

```
7 6 5 4 3 2 1 0   ◄──── Bit No.
X X X X X X X Y         Accumulator A contents
0 0 0 0 0 0 0 1         #1
─────────────
X X X X X X X 1         OR
```

Now in ell honesty, the program sequence we heve just described is e ridiculous wey of simulating flip-flop FFA end its three essocieted getes.

It is ridiculous because we simulated each gete es an Independent trensfer function. Insteed, let us consider the flip-flop, wlth its three gates, es e single transfer function. We cen represent the trensfer function with the following state definition:

Set Q̄ to 0 If RESET = 0, CH RDY = 1 end PW STROBE goes from 0 to 1. Set Q̄ to 1 otherwise.

How are we going to test for the transition of PW STROBE from 0 to 1?

Using interrupts, the test would be very simple; but we are not going to use interrupts until Chapter 5.

Without using interrupts, there is only one wey to check for e PW STROBE 0 to 1 trensition. We must input the contents of I/O Port B to Accumulator A, isolate bit 5, save the result, input the contents of I/O Port B to Accumulator A again, isolate bit 5 again, then compare the two bits for an old value of 0 and a new

value of 1. But this scheme is risky; it will only catch signal transitions which are lucky enough to occur in between the two instructions which load I/O Port B contents to Accumulator A:

PW STROBE

Ⓐ Ⓑ   Missed!

Ⓐ Ⓑ   Hit!
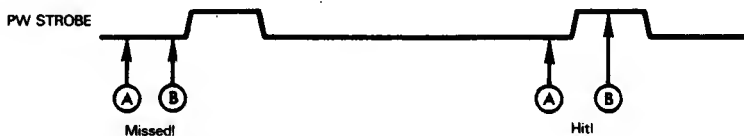
Ⓐ represents execution of first LDA A $C001 instruction

Ⓑ represents execution of second LDA A $C001 instruction

Within the logic of a microcomputer program, however, we have no need to rely on signal transitions. Event sequences are determined by instruction execution sequence. The whole concept of timing on the leading or trailing edge of a

signal pulse has no meaning. Instead of using PW STROBE signal transitions, therefore, we will use PW STROBE signal levels. Flip-flop FFA can now be described with the following state definition:

Set $\bar{Q}$ to 0 if RESET equals 0, CH RDY equals 1 and PW STROBE equals 1. Set $\bar{Q}$ to 1 otherwise.

If you are a logic designer, you may be deeply troubled by the blithe way in which we simply replace edge triggering with level triggering. We can do this within a microcomputer system because microcomputer programming gives us an ex-

tra degree of freedom, as compared with digital logic design: The order in which you stuff logic components into a PC card has nothing to do with the sequence in which logical events occur. Logic sequence is going to be controlled by edge and level triggering. But the order in which you write assembly language instructions is the order in which the instructions will be executed.

To drive this point home, look at the following flowchart which represents the state definition for flip-flop FFA:
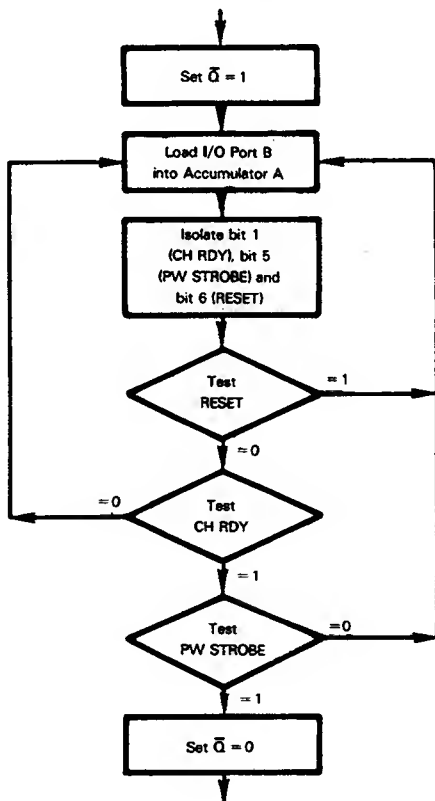
Again each rectangular box represents a data movement or manipulation operation, and each diamond represents logic which tests the condition of a status flag.

The order in which you write down instructions is the order in which instructions will be executed. With regard to the flowchart above, this execution sequence is represented by the continuous line of downward pointing arrows. Special Jump-On-Condition instructions allow the normal sequence to be modified, as represented by the horizontal arrows emanating from the sides of the diamonds. You can follow the arrows to the point where the Jump-On-Condition instruction takes you.

We will now rewrite the flip-flop FFA simulation treating the flip-flop and the three CLR logic gates as a single transfer function.

Since RESET, CH RDY and PW STROBE are all connected to pins of I/O Port B, we load the contents of I/O Port B into Accumulator A and isolate all three bits. Now there is only one combination of values that these three bits can have if a new print cycle is to begin. RESET must equal 0, while CH RDY and PW STROBE both equal 1. We will therefore redraw our program flowchart as follows:
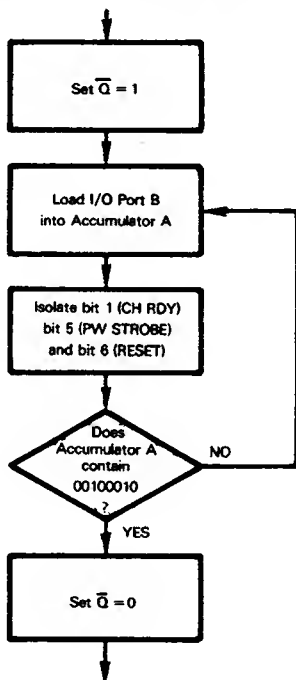
Our instruction sequence condenses to the following few instructions:

```
SIMULATION OF FFA AND ASSOCIATED LOGIC
        LDA  A    $E001      INITIALLY SET BIT 0 OF I/O PORT D TO 1
        ORA  A    #1
        STA  A    #$E001
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A
AND ISOLATE BITS 1, 5 AND 6 FOR CH RDY,
PW STROBE AND RESET, RESPECTIVELY
L10     LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$62       ISOLATE BITS 6, 5 AND 1
        CMP  A    #$22       IF RESET=0, CH RDY=1 AND
        BNE       L10        PW STROBE=1, NEW PRINT CYCLE STARTS
        LDA  A    $E001      OTHERWISE RETURN TO L10. START NEW
        AND  A    #$FE       PRINT CYCLE BY SETTING I/O PORT D, BIT 0 TO 0
        STA  A    $E001
NEW PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

The first three instructions in the above sequences simply set bit 0 of I/O Port D to 1. This is in anticipation of a new print cycle not beginning. Four instructions, beginning with the instruction labeled L10, are all that are needed to check for conditions which trigger the start of a new print cycle. These four instructions execute in 12 clock cycles which, assuming a 1 microsecond clock, means that PW STROBE must pulse high for at least 12 microseconds.

Providing RESET equals 0 while CH RDY and PW STROBE equal 1, a new print cycle must begin, so the last three instructions set bit 0 of I/O Port D to 0.

Our simulation of flip-flop FFA is complete.   •

FLIP-FLOP FFB$_W$

The next device in our logic sequence is another 7474 flip-flop, marked FFB$_W$ in Figure 3-1; it is just to the right of FFA$_W$. This flip-flop may be illustrated as follows:

3-38
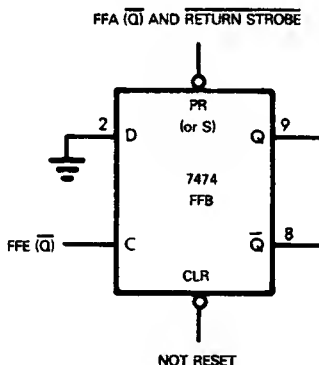
The following function table describes FFB, as wired above, with its D input tied to 0:

FFA (Q̄)	$\overline{\text{RETURN STROBE}}$	PRESET	NOT RESET (CLR)	FFE (Q̄) =CLOCK	Q	Q̄
0	0	0	1	X	1	0
0	1	0	0	X	unstable	
1	0	0				
1	1	1	0	X	0	1
		1	1	0 → 1	0	1

Chapter 2 provides the standard 7474 flip-flop function table; all we have done is remove the D column, and the rows that show D = 1. We can also remove the CLR column, and all rows that show CLR = 0, since CLR is tied to NOT RESET. NOT RESET will always be 1 within a print cycle, since FFA will not turn on if NOT RESET is 0.

The following simplified function table can now be used for FFB, assuming that CLR (NOT RESET) will always be 1 and D will always be 0:

FFA (Q̄) AND $\overline{\text{RETURN STROBE}}$ =PRESET	FFE (Q̄) =CLOCK	Q	Q̄
0	0 or 1	1	0
1	0 → 1	0	1

Let us take a look at the FFB PRESET input; it is FFA (Q̄) AND $\overline{\text{RETURN STROBE}}$.

3-39

RETURN STROBE, recall, is a signal input by external logic to initiate a
special print cycle which moves the printwheel back to its position of
visibility, but does not fire the printhammer or print a character. We
call this a "Printwheel Repositioning" print cycle. In between print cy-
cles, therefore, RETURN STROBE must be input high.

**Since RETURN STROBE is input low as an alternative method of initiating a print
cycle, when simulating FFB, we are going to have to consider RETURN STROBE
two ways:**

1) As a contributor to the PRESET input.

2) As a signal which can initiate a print cycle, bypassing flip-flop FFA.

But first, let us define the condition of flip-flop FFB in between print cycles.

As we have just seen in our simulation of flip-flop FFA, the FFA ($\overline{Q}$) output is high until the begin-
ning of a print cycle, when $\overline{Q}$ goes low; the FFA ($\overline{Q}$) output is therefore high in between print cy-
cles. By definition, RETURN STROBE is high in between print cycles, since RETURN STROBE low
is used to initiate a printwheel repositioning print cycle. **Therefore, the FFB PRESET input
will be high in between print cycles:**
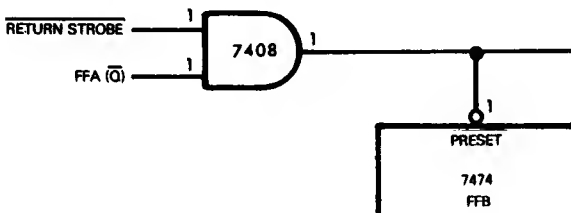
Since PRESET is input high in between print cycles, we are going to assume that at the beginning
of a print cycle FFB is off; that is, Q is output low and $\overline{Q}$ is output high. This also assumes that at
some recent time PRESET was input high when the $\overline{Q}$ output of flip-flop FFE went from 0 to 1.
As you will see later on, this is indeed what happens at the end of every print cycle.

**Coming into a new print cycle, therefore, FFB has a high PRESET input, with a
high $\overline{Q}$ output and a low Q output. This flip-flop now acts as a switch: it is turned
on by PRESET being input low; it is subsequently turned off by a clock 0 to 1 tran-
sition occurring after PRESET has again gone high:**
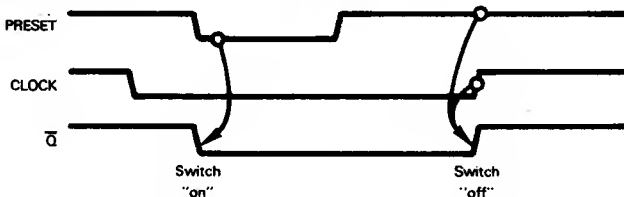
The switch "on" illustrated above occurs under two circumstances:

1) Immediately after the onset of a new print cycle, when FFA outputs $\overline{Q}$ low, thus forcing
    PRESET low.

2) When RETURN STROBE is input low signaling a printwheel repositioning print cycle.

The switch "off" occurs when the FFE ($\overline{Q}$) output makes a low to high transition while PRESET is being input high; this occurs at the end of every print cycle.

SIMULATING FLIP-FLOP FFB

Bit 1 of I/O Port D has been assigned to the $\overline{Q}$ output of flip-flop FFB. The switch "on" illustrated above is therefore simulated by the following three instructions:

<div style="float:right; border:1px solid black; padding:2px">SWITCHING
BITS ON</div>

```
LDA  A   $E001      LOAD FLIP-FLOP DATA BYTE
AND  A   #$FD       RESET BIT 1 TO 0
STA  A   $E001      RESTORE FLIP-FLOP DATA BYTE
```

This is how the AND instruction works:

```
7 6 5 4 3 2 1 0 ◄──── Bit No.
X X X X X X Y X      Accumulator A contents
1 1 1 1 1 1 0 1      #$FD
─────────────
X X X X X X 0 X      AND
```

<div style="float:right; border:1px solid black; padding:2px">SWITCHING
BITS OFF</div>

Subsequently the switch "off" will be simulated as follows:

```
LDA  A   $E001      LOAD FLIP-FLOP DATA BYTE
ORA  A   #2·        SET BIT 1 TO 1
STA  A   $E001      RESTORE FLIP-FLOP DATA BYTE
```

This is how the ORA instruction works:

```
7 6 5 4 3 2 1 0 ◄──── Bit No.
X X X X X X Y X      Accumulator A contents
0 0 0 0 0 0 1 0      #2
─────────────
X X X X X X 1 X      OR
```

We now encounter a situation where, with every best intention, we are not going to be able to directly simulate our digital logic.

It is easy enough to draw one 7474 flip-flop in a logic diagram and connect its pins to suitable signals. Having done that, you no longer need to worry about when a signal does, or does not change state. Unfortunately, an assembly language instruction sequence has no pins or signals; **assembly language will simulate events that are occurring at one instant in time only. For flip-flop FFB, this may be illustrated as follows:**
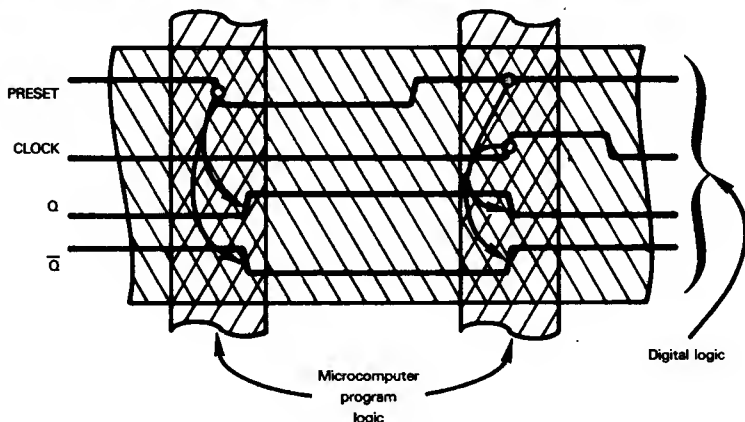
Immediately after flip-flop FFA turns on to usher in a new print cycle, it outputs $\overline{Q}$ low, which in turn switches flip-flop FFB on. FFB will not switch off until some point much later in the print cycle, when FFE outputs $\overline{Q}$ high. **We must therefore divide our simulation of FFB into two parts:**

1) At the beginning of our program we will simulate FFB switching on, since chronologically it is the next event within the print cycle.

2) Later on in the program, when we simulate FFE setting $\overline{Q}$ high, we must remember to simulate FFB switching off.

But that is not all there is to the FFB simulation. **We must also modify the instruction sequence that executes in between print cycles, so that RETURN STROBE input low can be simulated initiating a printwheel repositioning print cycle.**

With modified or new instructions shaded, this is how our program now looks:

```
IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BITS 1 AND 0 TO 1
        LDA  A    $E001      INPUT I/O PORT D TO ACCUMULATOR A
        ORA  A    #3         SET BITS 1 AND 0
        STA  A    $E001      RETURN RESULT
TEST FOR RETURN STROBE LOW
L10     LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$10       ISOLATE RETURN STROBE
        BEQ       FFB        IF IT IS 0, JUMP TO FFB SIMULATION
SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5 AND 6 FOR CH
RDY, PW STROBE AND RESET, RESPECTIVELY
        LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$62       ISOLATE BITS 6, 5 AND 1
        CMP  A    #$22       IF RESET=0, CH RDY=1 AND PW STROBE=1, START NEW
                             PRINT CYCLE
        BNE       L10        OTHERWISE RETURN TO L10
        LDA  A    $E001      TO START A NEW PRINT CYCLE,
        AND  A    #$FE       RESET I/O PORT D BIT 0 TO 0
        STA  A    $E001
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     LDA  A    $E001      LOAD I/O PORT D INTO ACCUMULATOR A
        AND  A    #$FD       RESET BIT 1 TO 0
        STA  A    $E001      RESTORE RESULT
```

We are not quite finished with our simulation of flip-flop FFB. Observe that the $\overline{Q}$ output from FFB goes to:

1) **A 7411 AND gate, located approximately at coordinate B6.**

2) **A 7432 OR gate, located at A7.**

The FFB (Q) output is not idle either, but we will look into it later.

First consider the 7411 AND gate located at B6.

If you refer back to the description of output signals, you will notice that CH RDY was declared to be high in between print cycles, but low during a print cycle.

In reality, CH RDY is output by the 7411 AND gate located at B6; therefore, in between print cycles, all three inputs to this AND gate must be high. Our analysis of flip-flop FFB shows that its $\overline{Q}$ output will indeed be high in between print cycles, but for the moment you must take it on faith that the other two signals input to the AND gate will also be high in between print cycles.

In any event, as soon as flip-flop FFB switches on, its $\overline{Q}$ output goes low, which means that no matter what the other two inputs to the 7411 AND gate do, CH RDY will also be driven low. This change in the status of CH RDY is simulated by adding the following instructions to our program:

```
TEST FOR RETURN STROBE LOW
L10     LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$10        ISOLATE RETURN STROBE
        BEQ     FFB         IF IT IS 0, JUMP TO FFB SIMULATION
SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5 AND 6
FOR CH RDY, PW STROBE AND RESET, RESPECTIVELY
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$62        ISOLATE BITS 6, 5 AND 1
        CMP A   #$22        IF RESET=0, CH RDY=1 AND PW STROBE=1, START NEW
        BNE     L10         PRINT CYCLE. OTHERWISE RETURN TO L10
        LDA A   $E001       TO START A NEW PRINT CYCLE,
        AND A   #$FE        SET I/O PORT D BIT 0 TO 0
        STA A   $E001
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     LDA A   $E001       LOAD I/O PORT D INTO ACCUMULATOR A
        AND A   #$FD        RESET BIT 1 TO 0
        STA A   $E001       RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FD        RESET BIT 1 TO 0
        STA A   $C001       RESTORE RESULT
```

We are now faced with an interesting problem. CH RDY becomes the D input to flip-flop FFA and it contributes to the CLR input of FFA. **What happens when CH RDY goes low in response to FFB switching on?**

Notice that PW STROBE only pulses high, therefore the OR gate located at coordinate B2 relies on CH RDY being high in order to provide a high input to the following AND gate. This AND gate, in turn, provides a high CLR input to flip-flop FFA. In other words, by the time flip-flop FFB turns "on" and switches CH RDY low, PW STROBE will have already gone low; thus inputs PW STROBE and CH RDY will both be low. **If you look back at flip-flop FFA's CLR truth table, you will find that when CH RDY and PW STROBE are both 0, CLR will always be 0.**

Therefore flip-flop FFA will switch off:
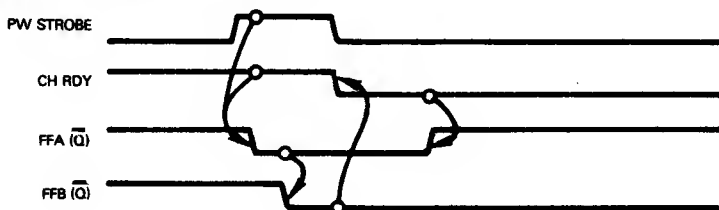
What does this mean? Our conclusion is that flip-flop FFA switches itself "on" at the beginning of a print cycle, but only stays on long enough to switch flip-flop FFB "on". When FFB turns "on", it sets CH RDY low, and that turns flip-flop FFA "off".

But here is the rub: if you look again at Figure 3-1, you will find that flip-flop FFA helps generate the J input to flip-flop FFC, in addition to switching on flip-flop FFB.

> **TIMING AND LOGIC SEQUENCE**

Now that events are serialized in time, we can go ahead and simulate flip-flop FFA being turned "off", so long as we remember, when simulating flip-flop FFC, that it receives $\overline{Q}$ low from flip-flop FFA. Bearing this precaution in mind, we will extend our program as follows:

```
TEST FOR RETURN STROBE LOW
L10     LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$10        ISOLATE RETURN STROBE
        BEQ     FFB         IF IT IS 0, JUMP TO FFB SIMULATION
SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5
AND 6 FOR CH RDY, PW STROBE AND RESET, RESPECTIVELY
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$62        ISOLATE BITS 6, 5 AND 1
        CMP A   #$22        IF RESET=0, CH RDY=1 AND PW STROBE=1, START NEW
        BNE     L10         PRINT CYCLE. OTHERWISE RETURN TO L10
        LDA A   $E001       TO START A NEW PRINT CYCLE,
        AND A   #$FE        SET I/O PORT D BIT 0 TO 0
        STA A   $E001
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     LDA A   $E001       LOAD I/O PORT D INTO ACCUMULATOR A
        AND A   #$FD        RESET BIT 1 TO 0
        STA A   $E001       RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FD        RESET BIT 1 TO 0
        STA A   $C001       RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
        LDA A   $E001       LOAD I/O PORT D TO ACCUMULATOR A
        ORA A   #1          SET BIT 0 TO 1
        STA A   $E001       RESTORE RESULT
```

Now look at the OR gate located at co-ordinate A7. This gate receives the FFB $\overline{Q}$ output as one of its inputs in order to generate PW REL. The other input to this OR gate is the AND of the O output from flip-flop FFF, plus the $\overline{Q}$ output of flip-flop FFD. You will find out shortly that these flip-flops are also turned "off" in between print cycles; they are turned on sequentially during the course of the print cycle. At the point where FFB switches on, FFF will be switched off, which means that its O output will be low; thus, the AND gate located at A6 will output low, which means that **OR gate 26 has been relying on the high $\overline{Q}$ output from FFB in order to output PW REL high:**
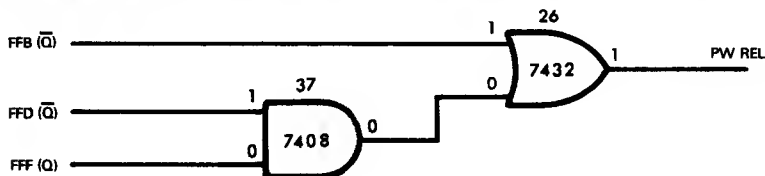
Now, when FFB switches "on" and outputs $\overline{Q}$ low, PW REL will also output low. We must therefore modify our program to output bits 0 and 1 of I/O Port B low, since both PW REL and CH RDY are going to be driven low. This is how our program now looks:

```
TEST FOR RETURN STROBE LOW
L10     LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$10        ISOLATE RETURN STROBE
        BEO     FFB         IF IT IS 0, JUMP TO FFB SIMULATION
SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5
AND 6 FOR CH RDY, PW STROBE AND RESET, RESPECTIVELY
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$62        ISOLATE BITS 6, 5 AND 1
        CMP A   #$22        IF RESET=0, CH RDY=1 AND PW STROBE=1, START
        BNE     L10         NEW PRINT CYCLE. OTHERWISE RETURN TO L10
        LDA A   $E001       TO START A NEW PRINT CYCLE,
        AND A   #$FE        SET I/O PORT D BIT 0 TO 0
        STA A   $E001
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     LDA A   $E001       LOAD I/O PORT D INTO ACCUMULATOR A
        AND A   #$FD        RESET BIT 1 TO 0
        STA A   $E001       RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE
SWITCHES PW REL LOW
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FC        RESET BITS 0 AND 1 TO 0
        STA A   $C001       RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
        LDA A   $E001       LOAD I/O PORT D TO ACCUMULATOR A
        ORA A   #1          SET BIT 0 TO 1
        STA A   $E001       RESTORE RESULT
```

Do we have to do anything about the Q output from flip-flop FFB? If you look at this output you will see that it ties directly to the RESET inputs of flip-flops FFC, FFD, and FFE. It also becomes one of the inputs to the 555 multivibrator.

In fact, the FFB O output is a clamping signal; when low, it shuts the four connected devices off; when high, these four devices are switched on.

The FFB Q output will be taken into account when we simulate the four devices connected to this signal. Therefore, our simulation of flip-flop FFB is done.

FLIP-FLOP FFC

This is the 74107 flip-flop at co-ordinate C2 in Figure 3-1. Since we are going to simulate four 74107 flip-flops, you should refer back to Chapter 2 if you cannot immediately recall the characteristics of this device.

Let us isolate flip-flop FFC to see how it works:
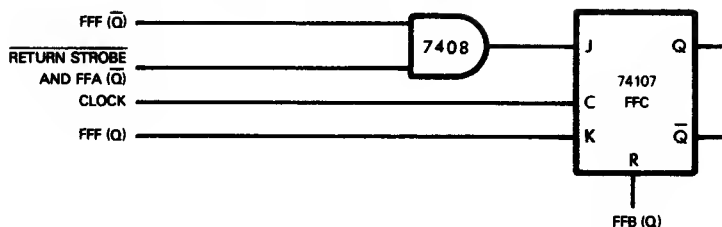
	INPUTS				
R	C	J	K	Q	Q̄
L	X	X	X	L	H
H	⊓	L	L	stay the same	
H	⊓	H	L	H	L
H	⊓	L	H	L	H
H	⊓	H	H	invert	

JK inputs here   become Q Q̄ outputs here
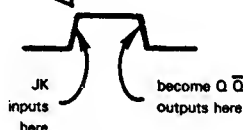
In between print cycles, the Q output of FFB, being low, switches flip-flop FFC off. FFC, therefore, outputs Q low and Q̄ high.

What happens when FFB is switched on depends on the J and K inputs arriving at FFC.

In between print cycles flip-flop FFF is switched off, therefore its Q output will be low. FFC receives its K input from the FFF Q output, therefore when FFC switches on, its K input will be 0.

The J input to FFC is generated as follows:
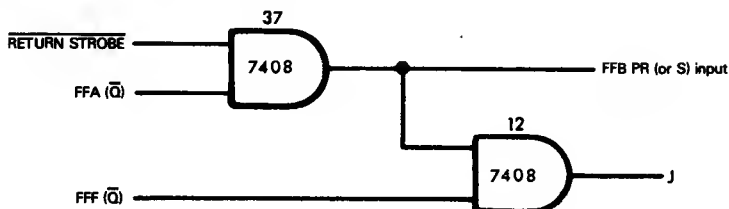
FFF (Q̄) will be high, since FFF is switched off. The FFC J input will therefore be identical to the FFB PR input, which we have already described.

In summary, this is the signal sequence which turns FFC on:
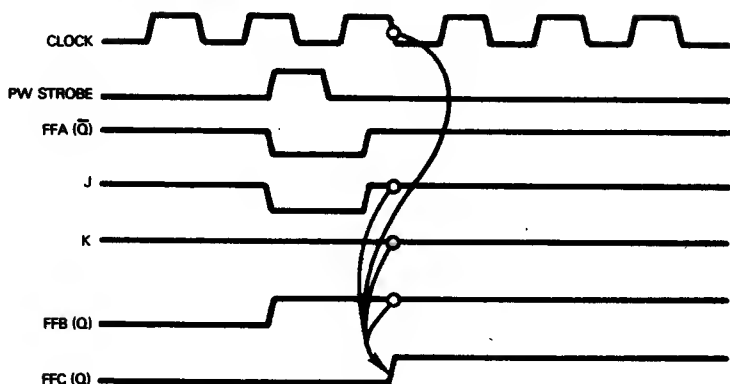
When the FFB Q output goes high, unclamping FFC, FFC waits until the FFA $\overline{Q}$ output goes high again; then FFC will receive a high input at J and a low input at K. On the trailing edge of the next clock pulse input to FFC, O will be output high and $\overline{Q}$ will be output low.

FFC waits for the FFA $\overline{O}$ output to go high again, because while FFA is switched on, $\overline{Q}$ is output low. While FFA ($\overline{Q}$) (or RETURN STROBE) is pulsed low, FFC receives a low J input. So long as FFC is receiving low J and K inputs, its outputs will not change — that is one of the properties of a 74107 flip-flop.

Flip-flop FFC will remain in its "on" state until some later point in the print cycle when flip-flop FFF switches on. At that time, flip-flop FFC wiii receive a high input at K and a low input at J; and that wiil cause FFC to switch off.

SIMULATING FLIP-FLOP FFC .

The simulation of flip-flop FFC is indeed straightforward; It invoives these three steps:

1) **We must adjust our initialization instructions to ensure that flip-flop FFC is reported as "off" in between print cycles.**

2) **The flip-flop FFB simulation must be followed immediately by instructions which simulate flip-flop FFC turning on.**

3) **We must remember to simulate FFC turning off — but that will not happen until some later point in the program.**

Now the following modifications to the beginning of our program insure that flip-flop FFC is simulated "off" in between print cycles:

```
IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BIT 2 TO 0, BITS 1 AND 0 TO 1
       LDA  A   $E001      INPUT I/O PORT D TO ACCUMULATOR A
 (A)   ORA  A   #3         SET BITS 1 AND 0
       AND  A   #$FB       RESET BIT 2
       STA  A   $E001      RETURN RESULT
TEST FOR RETURN STROBE LOW
L10    LDA. A   $C001      INPUT I/O PORT B TO ACCUMULATOR A
       AND  A   #$10       ISOLATE RETURN STROBE
       BEQ      FFB        IF IT IS 0, JUMP TO FFB SIMULATION
```

All we have done is add the AND instruction to reset I/O Port D bit 2 to 0:

Accumulator A
Contents

7 6 5 4 3 2 1 0 ◄── Bit No.

LDA A	$E001	X X X X X X X X
ORA A	#3	0 0 0 0 0 0 1 1
		X X X X X X 1 1
AND A	#$FB	1 1 1 1 1 0 1 1
		X X X X X 0 1 1

Recall that I/O Port D bit 2 has been assigned to flip-flop FFC.

What about the time delay that separates flip-flops B and C switching on? Recall that flip-flop FFC will not switch on until after flip-flop FFB has switched flip-flop FFA off. If this is a printwheel repositioning print cycle, then FFC will not switch on until RETURN STROBE is input high again.

The simplicity or complexity of our timing problem depends entirely on logic beyond Figure 3-1. There is nothing within the logic of Figure 3-1 that demands a time delay of fixed duration or, for that matter, any time delay separating FFB and FFC switching on. We will therefore pay no attention to the timing considerations associated with FFC switching on; rather we will simply add simulation to the end of our program as follows:

```
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     LDA  A   $E001        LOAD I/O PORT D INTO ACCUMULATOR A
        AND  A   #$FD         RESET BIT 1 TO 0
        STA  A   $E001        RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE SWITCHES
PW REL LOW
        LDA  A   $C001        INPUT I/O PORT B TO ACCUMULATOR A
        AND  A   #$FC         RESET BITS 0 AND 1 TO 0
        STA  A   $C001        RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
        LDA  A   $E001        LOAD I/O PORT D TO ACCUMULATOR A
        ORA  A   #1           SET BIT 0 TO 1
        STA  A   $E001        RESTORE RESULT
SIMULATE 74107 FLIP-FLOP FFC SWITCHING ON. SET BIT 2 OF I/O PORT D TO 1
        LDA  A   $E001        LOAD I/O PORT D INTO ACCUMULATOR A
        ORA  A   #4           SET BIT 2 TO 1
        STA  A   $E001        RESTORE RESULT
```

Ⓑ

If you are beginning to think like a programmer, you will detect an opportunity for economy in the simulation of flip-flop FFC switching on. **Observe that the three instructions directly above Ⓑ are also**

setting a bit of I/O Port D to 1. This generates the following sequence of events:

```
Input to Accumulator A                              Input to Accumulator A
   LDA  A    $E001    XXXXXXXX    LDA  A   $E001     XXXXXXX1
   ORA  A    #1        00000001    ORA  A   #4        00000100
   STA  A    $E001    XXXXXXX1    STA  A   $E001     XXXXX1X1
                       Output to I/O                    Output to I/O
                       Port D                           Port D
```
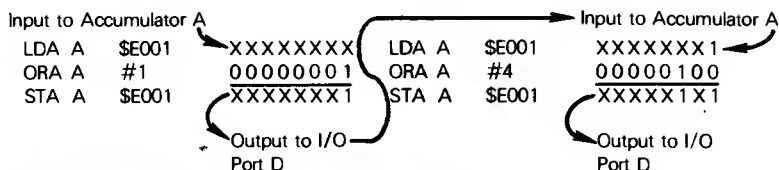
We can combine the two operations as follows:

```
   LDA  A    $E001    XXXXXXXX
   ORA  A    #5        00000101
                       XXXXX1X1
```

The instructions marked ⑧ now disappear, and are replaced by these modifications, marked ⓒ :

NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB   LDA A $E001   LOAD I/O PORT D INTO ACCUMULATOR A
        AND A   #$FD   RESET BIT 1 TO 0
        STA A $E001   RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE SWITCHES
PW REL LOW
        LDA A $C001   INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FC   RESET BITS 0 AND 1 TO 0
        STA A $C001   RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT D TO 1
    ⓒ LDA A $E001   LOAD I/O PORT D TO ACCUMULATOR A
        ORA A   #5   SET BITS 2 AND 0 TO 1
        STA A $E001   RESTORE RESULT

Our simulation of flip-flop FFC is now complete.

But before we continue, there is another programming economy worth exploring. I/O Port D outputs signals only, yet we load I/O Port D contents into Accumulator A prior to every signal level change. **If we load I/O Port D contents into Accumulator B, and use Accumulator B in no other way, then we can eliminate all instructions that load I/O Port D contents into Accumulator B — except for the first such instruction.** Since Accumulator B now serves as a buffer for I/O Port D, it will always have the same contents as I/O Port D; so why waste time loading identical data into Accumulator B?

> **ACCUMULATOR EFFECTIVE UTILIZATION**

Our program loses three instructions and changes as follows:

IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BIT 2 TO 0, BITS 1 AND 0 TO 1

```
          LDA B    $E001    INPUT I/O PORT D TO ACCUMULATOR B
          ORA B    #3       SET BITS 1 AND 0
          AND B    #$FB     RESET BIT 2
          STA B    $E001    RETURN RESULT
```

TEST FOR RETURN STROBE LOW

```
L10       LDA A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
          AND A    #$10     ISOLATE RETURN STROBE
          BEQ      FFB      IF IT IS 0, JUMP TO FFB SIMULATION
```

SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5 AND
6 FOR CH RDY, PW STROBE AND RESET, RESPECTIVELY

```
          LDA A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
          AND A    #$62     ISOLATE BITS 6, 5 AND 1
          CMP A    #$22     IF RESET=0, CH RDY=1 AND PW STROBE=1, START
          BNE      L10      NEW PRINT CYCLE. OTHERWISE RETURN TO L10
          AND B    #$FE     SET I/O PORT D BIT 0 TO 0 TO START NEW PRINT CYCLE
          STA B    $E001
```

NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON

```
FFB       AND B    #$FD     RESET BIT 1 TO 0
          STA B    $E001    RESTORE RESULT
```

SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE
SWITCHES PW REL LOW

```
          LDA A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
          AND A    #$FC     RESET BITS 0 AND 1 TO 0
          STA A    $C001    RESTORE RESULT
```

CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT D TO 1

```
          ORA B    #5       SET BITS 2 AND 0 TO 1
          STA B    $E001    RESTORE RESULT
```

START RIBBON MOTION PULSE SIMULATION

Recall that early in a print cycle the START RIBBON MOTION output signal is pulsed high to trigger external logic which advances the ribbon; thus when the printhammer fires, fresh ribbon is in front of the character being printed. The START RIBBON MOTION signal is generated by a 7411 AND gate (number 7) located at co-ordinate C7 in Figure 3-1. This AND gate has three inputs:

1) $\overline{\text{HAMMER ENABLE FF}}$. This is a signal input to identify a printwheel repositioning print cycle.

2) The O output from flip-flop FFC.

3) The $\overline{Q}$ output from flip-flop FFD.

$\overline{\text{HAMMER ENABLE FF}}$ will be high unless a printwheel repositiong print cycle is in progress, in which case the ribbon does not have to be moved. This signal, therefore, suppresses the START RIBBON MOTION pulse.

In between print cycles, flip-flops FFC and FFD are both switched off; therefore FFC (O) is low and FFD ($\overline{Q}$) is high. **The FFC (Q) output holds the START RIBBON MOTION signal low.**

When FFC switches on during a normal print cycle, all inputs to AND gate 7 will be high, so START RIBBON MOTION will pulse high; it will stay high until flip-flop FFD switches on, at which time FFD will output Q̄ low; and that will drop START RIB-BON MOTION pulse low. **Timing may be illustrated as follows:**
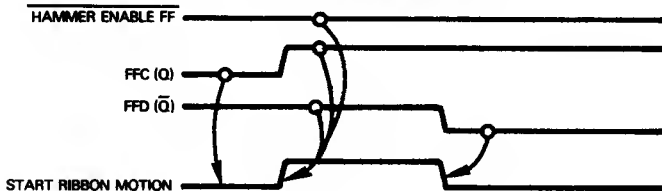
If you look at the timing diagram illustrated in Figure 3-2, you will see that the START RIBBON MOTION output pulse is extremely short. Therefore, instead of using flip-flop FFD to time the end of the START RIBBON MOTION HIGH PULSE, **we will simply execute instructions to turn bit 3 of I/O Port B on, then immediately turn it off, as follows:**

```
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     AND B   #$FD      RESET BIT 1 TO 0
        STA B   $E001     RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE
SWITCHES PW REL LOW
        LDA A   $C001     INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FC      RESET BITS 0 AND 1 TO 0
        STA A   $C001     RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT D TO 1
        ORA B   #5        SET BITS 2 AND 0 TO 1
        STA B   $E001     RESTORE RESULT
PULSE START RIBBON MOTION HIGH
        ORA A   #8        SET BIT 3 HIGH
        STA A   $C001     OUTPUT TO I/O PORT B
        AND A   #$F7      SET BIT 3 LOW
        STA A   $C001     OUTPUT TO I/O PORT B
```

Observe that we do not have to load I/O Port B contents into Accumulator A; the required data is still in Accumulator A following simulation of CH RDY and PW REL switching low.

We can calculate the START RIB MOTION pulse width by adding the instruction execution times between pin 3 of I/O Port B being set high, then being reset low:

| PULSE WIDTH |
| CALCULATION |

Cycles	Instruction		
5	STA A	$C001	OUTPUT TO I/O PORT B
2	AND A	#$F7	SET BIT 3 LOW
5	STA A	$C001	OUTPUT TO I/O PORT B

Pulse width = 7 cycles, or 7 microseconds using a 1 microsecond clock.

What happens next? Our logic sequence may take us to flip-flop FFD, to the right of FFC, or we may drop down to the 74121 one-shot number 36, just below and to the right of FFC.

One-shot 36 has its two A inputs tied to ground, which means that they will both input low. If you look at the 74121 function table given in Chapter 2, you will find that in this configuration, a one-shot output is triggered by a low-to-high transition at B. FFC ($\overline{O}$) provides this trigger. Any other B input will keep this one-shot turned off — which means that **Q and $\overline{Q}$ will output low and high, respectively, until much later in the print cycle, when FFC switches off;** that is when the FFC $\overline{Q}$ output makes a low-to-high transition.

Flip-flop FFD becomes the next device to be simulated.

FLIP-FLOP FFD

Flip-flop FFD receives its J input directly from the FFC (O) output; it receives its K input from the FFC ($\overline{O}$) output. Remember, since one-shot 36 is still switched off, its $\overline{O}$ output will be high; that means AND gate 12 will simply allow the FFC ($\overline{O}$) output to propagate straight through, to become the FFD (K) input.

Now, flip-flop FFD receives the same reset and clock signals as FFC, therefore **flip-flop FFD will simply switch on one clock cycle later than flip-flop FFC.**

SIMULATING FLIP-FLOP FFD

The simulation of flip-flop FFD is almost identical to the simulation of flip-flop FFC; the principal difference is that bit 3 of I/O Port D has been assigned to flip-flop FFD. Once again, we are going to limit ourselves to switching flip-flop FFD on and ensuring that its setting in between print cycles is correct.

Flip-flop FFD is switched off later in the print cycle; we must therefore remember to switch it off later in the program.

Here are the necessary program modifications and additions:

```
IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BITS 3 AND 2 TO 0, BITS 1 AND 0 TO 1
        LDA B   $E001       INPUT I/O PORT D TO ACCUMULATOR B
        ORA B   #3          SET BITS 1 AND 0
        AND B   #$F3        RESET BITS 3 AND 2
        STA B   $E001       RETURN RESULT
TEST FOR RETURN STROBE LOW
L10     LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$10        ISOLATE RETURN STROBE
        BEO     FFB         IF IT IS 0, JUMP TO FFB SIMULATION
        ≡                   ≡
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT D TO 1
        ORA B   #5          SET BITS 2 AND 0 TO 1
        STA B   $E001       RESTORE RESULT
PULSE START RIBBON MOTION HIGH
        ORA A   #8          SET BIT 3 HIGH
        STA A   $C001       OUTPUT TO I/O PORT B
        AND A   #$F7        SET BIT 3 LOW
        STA A   $C001       OUTPUT TO I/O PORT B
SIMULATE FFD TURNING ON. SET BIT 3 OF I/O PORT D TO 1
        ORA B   #8          SET BIT 3 TO 1
        STA B   $E001       RESTORE RESULT
```

Note that we do not have to load I/O Port D contents into Accumulator B; the correct data is already there.

If the program modifications and additions illustrated above are not immediately obvious, compare them to the flip-flop C simulation. Do not go on if you do not understand the flip-flop FFD program changes.

Just as the simulation of FFC switching on ⒷB was absorbed into the FFB simulation (Ⓒ), so **the simulation of FFD switching on (Ⓔ) can be absorbed as follows:**

```
NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB     AND B    #$FD      RESET BIT 1 TO 0
        STA B    $E001     RESTORE RESULT
SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE
SWITCHES PW REL LOW
        LDA A    $C001     INPUT I/O PORT B TO ACCUMULATOR A
        AND A    #$FC      RESET BITS 0 AND 1 TO 0
        STA A    $C001     RESTORE RESULT
CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC AND FFD TURNING ON. SET BIT 2 OF I/O PORT D TO 1
        ORA B    #$0D      SET BITS 3, 2 AND 0 TO 1                    ◄── Ⓕ
        STA B    $E001     RESTORE RESULT
PULSE START RIBBON MOTION HIGH
        ORA A    #8        SET BIT 3 HIGH
        STA A    $C001     OUTPUT TO I/O PORT B
        AND A    #$F7      SET BIT 3 LOW
        STA A    $C001     OUTPUT TO I/O PORT B
```

If the simulations are combined (Ⓕ), flip-flops FFC and FFD will switch on at exactly the same instant in time.

The logic in Figure 3-1 shows FFD switching on one clock pulse after FFC. If the clock period is two microseconds, then there will be a two microsecond delay between flip-flops FFD and FFC switching on. Both our simulations are wrong.

Does this matter? We honestly cannot tell with the informa-tion at hand. We do not know how external logic uses the FFC and FFD outputs. **If the switching time interval between these two flip-flops has to be very close to two microseconds, then our**

simulation is not going to work. Either the two flip-flops must become part of "external logic", or some other means of simulating the eventual overall function must be found.

If external logic demands some switching time delay, but is not fussy about the length of the time delay, then our simulation of flip-flop FFD is adequate.

It is quite possible that the logic in Figure 3-1 shows a switching time delay bet-ween flip-flops FFC and FFD only to define the leading and trailing edges of the START RIBBON MOTION pulse; but we have taken care of this high pulse by se-quentially executing instructions that output 1, then 0 to bit 3 of I/O Port B. So far as logic internal to Figure 3-1 is concerned, therefore, the need for a switching time delay bet-ween flip-flops FFC and FFD disappears. This being the case, **we will assume that external logic has no need for a switching time delay between flip-flops FFC and FFD; and we will adopt the shorter, combined simulation identified by** Ⓕ

FLIP-FLOP FFE

The next device in our logic sequence is flip-flop FFE. The circuitry surrounding this flip-flop is almost identical to FFD.

The FFE (K) input is tied to the FFD (Q̄) output, switched by another component of AND gate 12. The other input to this AND gate is the Q̄ output of one-shot 49. One-shot 49 is wired in the same way as one-shot 36, which we have just described.

The transition of flip-flop FFD's $\overline{Q}$ output from 0 to 1 will occur when FFD is switched off; and this is the transition which will trigger one-shot 49. Therefore, **one-shot 49 will output $\overline{Q}$ high until flip-flop FFD is switched off, which means that when FFD switches on, its $\overline{Q}$ output will propagate straight through the AND gate connecting it to the FFE (K) input:**
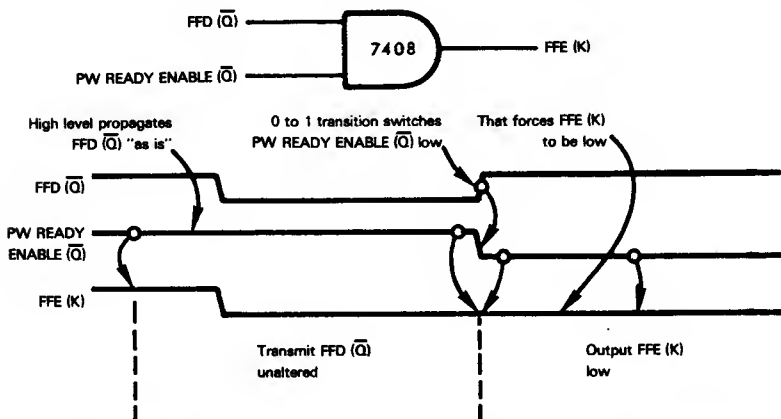
The **unique feature of flip-flop FFE is the way in which its J input is generated.** This input is the AND of the FFD (Q) output and input signal FFI. Now, the Q output of FFD will go high as soon as FFD switches on; but **FFI is input low from the beginning of the print cycle until the printwheel has correctly positioned itself.** (We described the function of this input signal earlier in the chapter.) **The timing associated with FFI may be illustrated as follows:**
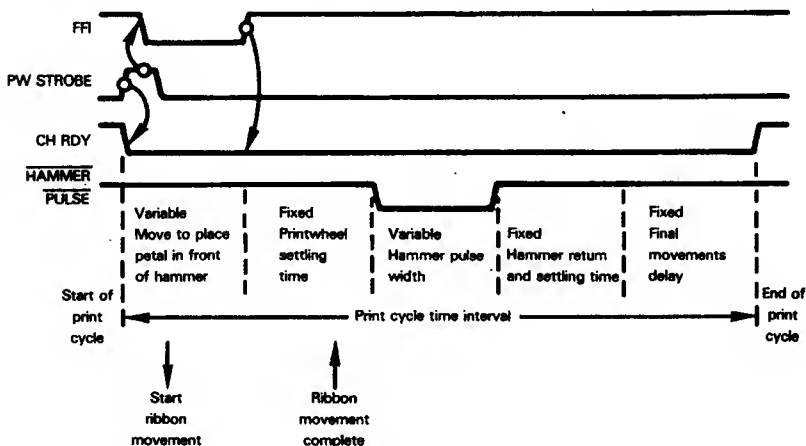
So long as FFI is low, flip-flop FFE will receive a low J input; low J and K inputs, you will recall, hold the O outputs of a 74107 flip-flop in their prior condition. Thus **input signal FFI has been used to create the first time delay of the print cycle: a variable time delay needed to move the required printwheel petal in front of the printhammer. Simulating this time delay is simple enough; it may be illustrated as follows:**

```
PULSE START RIBBON MOTION HIGH
        ORA  A    #8          SET BIT 3 HIGH
        STA  A    $C001       OUTPUT TO I/O PORT B
        AND  A    #$F7        SET BIT 3 LOW
        STA  A    $C001       OUTPUT TO I/O PORT B
TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    LDA  A    $C000       INPUT I/O PORT A TO ACCUMULATOR A
        ROL  A                SHIFT BIT 7 INTO CARRY
        BCC       VLDC        STAY IN LOOP IF CARRY IS ZERO
AT END OF DELAY SIMULATE FFE SWITCHING ON
        AND  B    #$DF        RESET BIT 5
        ORA  B    #$10        SET BIT 4
        STA  B    $E001       OUTPUT THE RESULT
```

In order to generate the initial time delay, we simply execute a continuous program loop which inputs the contents of I/O Port A to Accumulator A. Remember, we have reserved Accumulator B to hold the current bit values for I/O Port D; all other data uses Accumulator A. Bit 7 of I/O Port A has been assigned to input signal FFI. We test this bit by shifting it into the Carry status. If the Carry status then has a 0 content, FFI must still be low; so we stay within the loop. As soon as a 1 is shifted into the Carry

<table>
<tr><td>TIME DELAY
OF VARIABLE
LENGTH</td></tr>
<tr><td>JUMP ON
NO CARRY</td></tr>
</table>

status, the BCC instruction will create a "false" result; the next sequential instruction executes and we are out of the time delay loop:

```
        ┌─VLDC    LDA  A    $C000
C = 0 ──┤         ROL  A
        └──────BCC          VLDC
   C = 1 ──AND  B    #$DF
```

Branch if Carry Clear means branch if Carry is 0 (clear). "Branch" means "do not go on to the next sequential instruction", instead go to VLDC.

Tha last four instructions of tha FFE simulation show both outputs of this flip-flop becoming output signals. This meets requirements of Figura 3-1. We therefore reset bit 5 (it represents the Q̄ output) and we set bit 4 (it represents the O input).

The instruction sequence executad in batween print cyclas will have to ba modified to ensure that bit 5 has initially been set to 1, while bit 4 has initially been reset to 0. Here are the required modifications:

```
IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BITS 4, 3 AND 1 TO 0, BITS 5, 1 AND 0 TO 1
        LDA  B    $E001       INPUT I/O PORT D TO ACCUMULATOR B
        ORA  B    #$23        SET BITS 5, 1, AND 0 TO 1
        AND  B    #$E3        RESET BITS 4, 3, AND 2 TO 0
        STA  B    $E001       RETURN RESULT
TEST FOR RETURN STROBE LOW
L10     LDA  A    $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$10        ISOLATE RETURN STROBE
        BEQ       FFB         IF IT IS 0, JUMP TO FFB SIMULATION
```

PW SETTLING ONE-SHOT

The PW SETTLING one-shot is the 74121 device at co-ordinate B5 in Figure 3-1.
We have described this device in Chapter 2. With its two A inputs tied to ground, **this one-shot is triggered by a low-to-high transition at its B input. Since the B input is tied to the FFE Q output, this transition occurs as soon as flip-flop FFE switches on.**

The PW SETTLING one-shot has a two millisecond delay. This delay results from the external capacitor/resistor combination marked C1 and R1. Therefore as soon as FFE switches on, the PW SETTLING one-shot outputs $\overline{Q}$ low for two milliseconds.
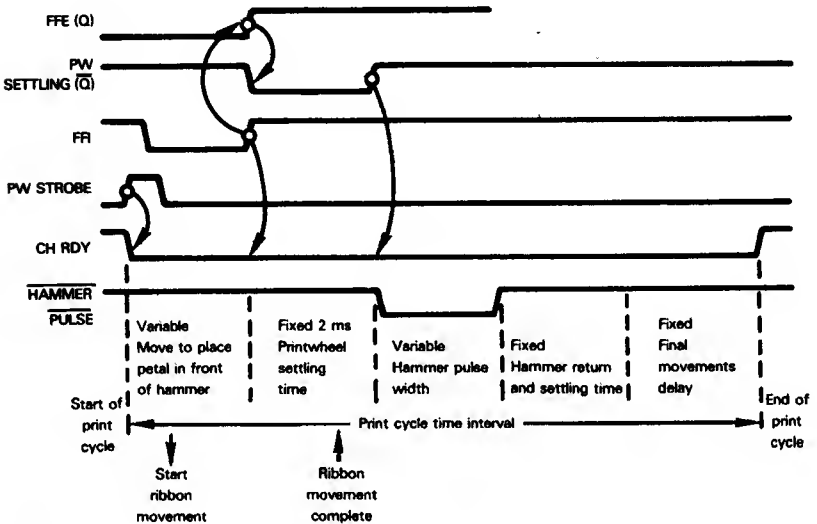
SIMULATING THE PW SETTLING ONE-SHOT

Simulating the one-shot time delay is simple enough and may be illustrated as follows:

> **ONE-SHOT
> TIME DELAY
> SIMULATION**

```
PULSE START RIBBON MOTION HIGH
        ORA A   #8          SET BIT 3 HIGH
        STA A   $C001       OUTPUT TO I/O PORT B
        AND A   #$F7        SET BIT 3 LOW
        STA A   $C001       OUTPUT TO I/O PORT B
TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    LDA A   $C000       INPUT I/O PORT A TO ACCUMULA-
                            TOR A
        ROL A               SHIFT BIT 7 INTO CARRY
        BCC     VLDC        STAY IN LOOP IF CARRY IS ZERO
AT END OF DELAY SIMULATE FFE SWITCHING ON
        AND B   #$DF        RESET BIT 5
        ORA B   #$10        SET BIT 4
        STA B   $E001       OUTPUT THE RESULT
SIMULATE 2 MS PW SETTLING TIME DELAY
        LDX     #$FA        LOAD INITIAL TIME DELAY CONS-
                            TANT
PWS     DEX                 DECREMENT INDEX REGISTER
        BNE     PWS         REDECREMENT IF NOT ZERO
```

There are two instructions in the time delay loop: DEX and BNE; thus the total time delay can be computed as follows:

$$250 \times 8 + 3 = 2003 \text{ microseconds}$$

Initial Index — Time to execute — Time to execute
register   DEX and BNE   initial LDX
contents   instructions   instruction

The above equation assumes a 1 microsecond clock period.

Notice that we are using the Index register which can hold a 16-bit initial value, even though the initial Index register contents is 250 — which could fit in an 8-bit Accumulator. Why do we do this? The reason is because the DEX instruction takes four machine cycles to execute, whereas a DEC, which decrements the contents of one of the Accumulators, executes in two machine cycles. Thus it would only require six machine cycles to execute the two-instruction loop were we to decrement an Accumulator. In this instance the initial Accumulator contents would have to be 333 — and that is a number which would not fit within the 8 bits of the Accumulator.

FLIP-FLOP FFF

Once the PW SETTLING one-shot has timed out, we are ready to fire the printhammer. The 555 multivibrator is actually going to generate the printhammer firing pulse, but it is most important to ensure that the printhammer does not fire while any part of the print or carriage mechanisms is moving. The 555 one-shot is therefore triggered by flip-flop FFF which, in turn, is switched on by a J input that is the AND of many safeguard signals. Let us isolate flip-flop FFF and examine its inputs.
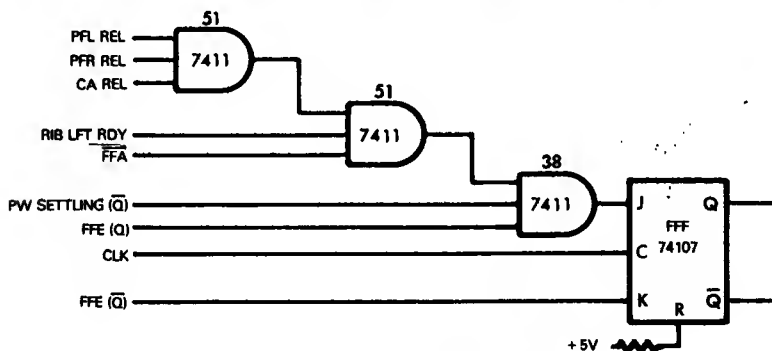
With its Clear (R) input tied to +5V, flip-flop FFF has the following function table:

INPUTS		OUTPUTS	
J	K	Q	Q̄
0	0	No change	
1	0	1	0
0	1	0	1
1	1	Complement	

Inputs at   Control outputs
positive clock   at negative clock
edge   edge
CLOCK

In between print cycles, FFE is "off", so the K input to FFF is high. The flip-flop FFF J input will be low since the FFE (Q) output will be low, and FFE (Q̄) is one contributor to FFF (J).

In between print cycles, therefore, flip-flop FFF is "off", since a low J input and a high K input generate steady outputs of Q=0, Q̄=1; this is characteristic of a flip-flop in its "off" condition.

Now when FFE switches on, it inputs a low K to FFF. So long as the J input is also low, no change occurs. As soon as the seven signals contributing to FFF (J) are all high, flip-flop FFF will receive a high J input; this will switch flip-flop FFF on — Q is then output high and Q̄ is output low.

SIMULATING FLIP-FLOP FFF

Coming out of the simulation of FFE, we know that FFE (Q) and FFE (Q̄) have correct levels for FFF to switch on.

Coming out of the simulation of the PW SETTLING one-shot, the one-shot Q̄ output must be high:
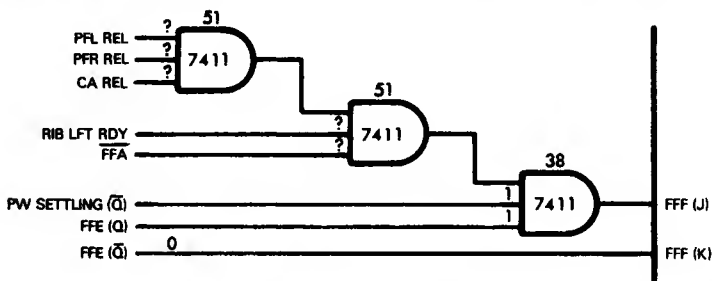
All that is needed is to test the five remaining interlock signals; as soon as they are all high, we simulate flip-flop FFF switching on. This is the instruction sequence:

```
TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC    LDA  A    $C000    INPUT I/O PORT A TO ACCUMULATOR A
        ROL  A             SHIFT BIT 7 INTO CARRY
        BCC       VLDC     STAY IN LOOP IF CARRY IS ZERO
AT END OF DELAY SIMULATE FFE SWITCHING ON
        AND  B    #$DF     RESET BIT 5
        ORA  B    #$10     SET BIT 4
        STA  B    $E001    OUTPUT THE RESULT
SIMULATE 2 MS PW SETTLING TIME DELAY
        LDX       #$FA     LOAD INITIAL TIME DELAY CONSTANT
PWS     DEX                DECREMENT INDEX REGISTER
        BNE       PWS      REDECREMENT IF NOT ZERO
SIMULATE FLIP-FLOP FFF SWITCHING ON
FFF     LDA  A    $C000    INPUT I/O PORT A CONTENTS TO ACCUMULATOR A
        COM  A             COMPLEMENT TO TEST FOR 1 BITS
        AND  A    #$1F     ISOLATE BITS 0 THROUGH 4
        BNE       FFF      IF ANY BITS ARE 1, STAY IN LOOP
        ORA  B    #$40     SET BIT 6 OF I/O PORT D TO 1
        STA  B    $E001
```

By now, you should be able to understand instructions as they are added to the program.

The first four instructions simply load the contents of I/O Port A into Accumulator A and test for 1s in the low order five bits. Until such time as all five bits are 1, the program will remain in the four-instruction loop that begins with LDA A $C000 and ends with BNE FFF.

When bits 0 through 4 all equal 1, the COM instruction changes all these bits to 0:

```
                        Accumulator A contents
FFF     LDA  A   $C000      X X X 1 1 1 1 1
        COM  A               X̄ X̄ X̄ 0 0 0 0 0
        AND  A   #$1F        0 0 0 1 1 1 1 1
                            ‾0‾0‾0‾0‾0‾0‾0‾0    Zero status = 1
        BNE      FFF      Return to FFF only if Zero status = 0
        ORA  B   #$40     Continue here if Zero status is 1
```

The BNE instruction no longer deflects program execution back to FFF, rather, it allows the next sequential instruction to be executed.

Observe that following the BNE FFF instruction we can perform an immediate OR upon the contents of Accumulator B in order to set the bit of I/O Port D which has been assigned to signal FFF. This is because we have reserved Accumulator B to serve as a storage for I/O Port D; and we use Accumulator B in no other way. Thus it requires just two instructions to simulate flip-flop FFF being switched on. The ORA instruction forces bit 6 of Accumulator B to 1, while leaving other bits of Accumulator B with their previous assignments. The STA instruction outputs the modified Accumulator B contents to I/O Port D.

We can make the final modification to the instruction sequence which correctly sets flip-flop status in between print cycles. This is what we finish up with:

IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BITS 6, 4, 3, AND 2 TO 0. BITS 5, 1 AND 0 to 1.
```
        LDA  B   $E001      INPUT I/O PORT D TO ACCUMULATOR B
        ORA  B   #$23       SET BITS 5, 1, AND 0 TO 1
        AND  B   #$A3       RESET BITS 6, 4, 3, AND 2 TO 0
        STA  B   $E001      RETURN RESULT
```

What happens when flip-flop FFF switches on?

The FFF (Q) output goes up to pin 9 of AND gate 37 at co-ordinate A6. This is part of the logic which contributes to the PW REL signal. However, **the transition of the FFF (Q) output from low-to-high is not significant,** since the other input to AND gate 37 is the FFD (Q̄) output which is currently low. The FFF (Q) output is connected to AND gate 37 to hold PW REL low early in the print cycle when FFD (Q̄) is high.

The FFF Q and Q̄ outputs contribute to the FFC J and K inputs. FFF (Q̄) is one contributor to AND gate 12, the output of which becomes the FFC (J) input. The other contributor to this AND gate is the output of AND gate 37 at co-ordinate A3, which is constantly high by this time in the print cycle; therefore, when the FFF (Q̄) output goes low the FFC (J) input also goes low. The K input to FFC is the FFF (Q) output. **FFC will therefore switch off when K goes high and that will not happen until FFF switches on.**

In our simulation, however, we are going to postpone FFC switching off until the end of HAMMER PULSE. This is because the purpose of FFC switching off is to trigger the PW RELEASE ENABLE one-shot, which creates the time delay needed by the printhammer to settle back. Thus instead of using parallel delays:

PW RELEASE ENABLE ONE-SHOT
(Fixed Delay)

HAMMER
FIRING
PULSE
(Variable Delay)

EFFECTIVE PW RELEASE
ENABLE DELAY
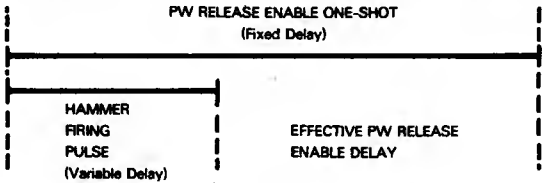
we will implement serial delays, which more immediately meet logic needs:

HAMMER
FIRING
PULSE
(Variable Delay)

PW RELEASE ENABLE
ONE-SHOT (Fixed Delay)
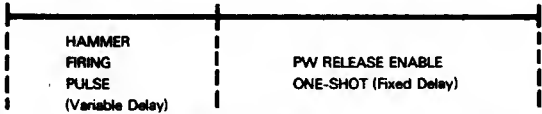
The hammer firing pulse is generated by the 555 one-shot. Therefore the 555 one-shot provides the next event in our chronological sequence; it is triggered by a high-to-low transition at pin 2. This pin is created as follows:

FFC (Q)
FFF (Q)
44
7408
43
7404
2
555
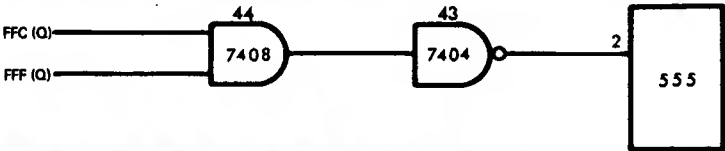
This is the sequence of events that must be simulated:

555 (Q̄)

FFF (Q)

PW
SETTLING (Q̄)

FFI

PW STROBE

CH RDY

HAMMER
PULSE

| Variable Move to place petal in front of hammer | Fixed 2 ms Printwheel settling time | Variable Hammer pulse width | Fixed Hammer return and settling time | Fixed Final movements delay |

Start of
print cycle

Print cycle time interval

End of
print cycle

Start ribbon
movement

Ribbon movement
complete
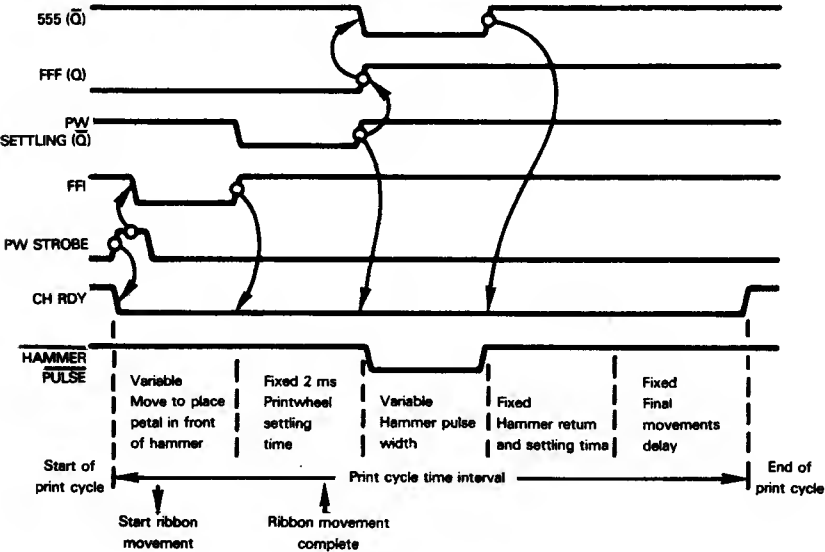
THE 555 MULTIVIBRATOR

Compare the way in which the 555 multivibrator has been wired in Figure 3-1 with the description of the multivibrator, as given in Chapter 2; you will see that **flip-flop FFB switches the multivibrator "off" in between print cycles** by inputting a low reset at pin 4. **The flip-flop FFF (Q) output triggers the multivibrator,** as we have just described.

The duration of the one-shot output pulse is controlled by inputs H1 through H6. One of these six inputs will be true while the other five will be false; thus the multivibrator, once triggered, will output a one-shot which can have a "high" pulse with one of six possible durations.

> **ONE-SHOT VARIABLE PULSE**

The 555 multivibrator one-shot output is eventually inverted to become a hammer pulse output; however, for the hammer pulse output to occur, additional inputs to AND gates 37 and 38, located at co-ordinates B8 and C7, respectively, must also be high. We may represent the hammer pulse logic as follows:
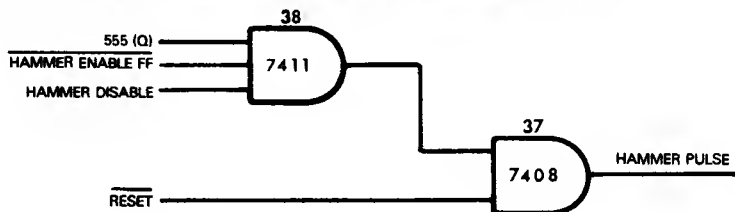
We will simply have to test the $\overline{\text{HAMMER ENABLE FF}}$ input before generating a HAMMER PULSE output.

The HAMMER DISABLE switch must be simulated.

RESET we can ignore, since RESET logic is being simulated in between print cycles.

SIMULATING MULTIVIBRATOR 555

The simulation of the 555 multivibrator consists of the following logic sequence:

1) **Determine if conditions have been satisfied for a 555 one-shot output to be transmitted as a HAMMER PULSE output.**

2) **Examine inputs H1 through H6. Based on these inputs, create one of six possible time delays.**

3) **If conditions for a HAMMER PULSE output have been satisfied, translate the 555 one-shot output into a HAMMER PULSE output.**

Let us first look at the HAMMER PULSE output enabling logic. Testing the condition of $\overline{\text{HAMMER ENABLE FF}}$ is simple enough, it has been assigned pin 6 of I/O Port A.

But there are no switches in assembly language programs; how are we going to simulate the hammer disable? We could assign the one remaining pin — pin 5 of I/O Port A to an input signal generated by an external switch. It would be just as simple to place this switch in the path of $\overline{\text{HAMMER ENABLE FF}}$ as follows:

> **LOGIC EXCLUDED FROM MICROCOMPUTER**
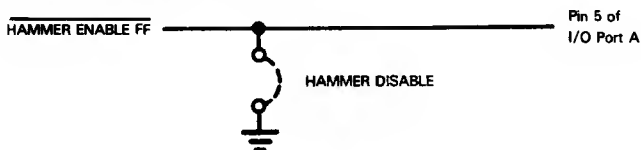
We will therefore ignore the hammer disable switch and enable a hammer pulse output providing the HAMMER ENABLE FF input is high.

What about the six possible durations for the 555 multivibrator output? We described in Chapter 2 how a time delay can be created by loading a 16-bit value into the Index register, then decrementing this register within a program loop, remaining in the program loop until a decrement to zero occurs. **Selecting one of six possible time delays is as simple as selecting one of six possible initial time constants. We can now simulate our 555 multivibrator as follows:**

```
SIMULATE 2 MS PW SETTLING TIME DELAY
        LDX     #$FA      LOAD INITIAL TIME DELAY CONSTANT
PWS     DEX               DECREMENT INDEX REGISTER
        BNE     PWS       REDECREMENT IF NOT ZERO
SIMULATE FLIP-FLOP FFF SWITCHING ON
FFF     LDA A   $C000     INPUT I/O PORT A CONTENTS TO ACCUMULATOR A
        COM A             COMPLEMENT TO TEST FOR 1 BITS
        AND A   #$1F      ISOLATE BITS 0 THROUGH 4
        BNE     FFF       IF ANY BITS ARE 1, STAY IN LOOP
        ORA B   #$40      SET BIT 6 OF I/O PORT D TO 1
        STA B   $E001
TEST HAMMER ENABLE FF
        LDA A   $C000     INPUT I/O PORT A TO ACCUMULATOR A
        AND A   #$40      ISOLATE BIT 6
        BEQ     HP0       IF ZERO, BYPASS SETTING HAMMER PULSE LOW
HAMMER ENABLE FF IS HIGH, SO HAMMER PULSE MUST BE OUTPUT LOW,
THEREFORE SET BIT 2 OF I/O PORT B TO 0
        LDA A   $C001     INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$FB      SET BIT 2 TO 0
        STA A   $C001     OUTPUT RESULT
COMPUTE TIME DELAY
HP0     LDX     #DELY     LOAD DELAY BASE ADDRESS INTO INDEX REGISTER
        LDA A   H1H6      LOAD SELECTOR INTO ACCUMULATOR A
HP1     LSR A             SHIFT ACCUMULATOR A RIGHT
        INX               INCREMENT INDEX REGISTER BY 2
        INX
        BCC     HP1       IF CARRY IS CLEAR ROTATE AND INCREMENT AGAIN
        LDX     0,X       LOAD 16-BIT DELAY COUNTER INTO INDEX REGISTER
TDLY    DEX               EXECUTE TIME DELAY LOOP
        BNE     TDLY
OUTPUT HAMMER PULSE HIGH AGAIN
        LDA A   $C001     INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #4        SET BIT 2 TO 1
        STA A   $C001     OUTPUT RESULT
```

Compared to the other devices we have simulated thus far, the 555 multivibrator requires a lot of simulation instructions. While it may look as though there is a lot to understand, the logic is, in fact, quite simple; so let us take it one piece at a time.

Initially we test HAMMER ENABLE FF. HAMMER PULSE will be output low only if HAMMER ENABLE FF is high. The three instructions which test the status of HAMMER ENABLE FF are:

```
| SIGNAL |
| ENABLE |
```

```
        LDA A   $C000     INPUT I/O PORT A TO ACCUMULATOR A
        AND A   #$40      ISOLATE BIT 6
        BEO     HP0       IF ZERO, BYPASS SETTING HAMMER PULSE LOW
```

There are two aspects of these three instructions which need to be explained. First, there is the logic being implemented. We are determining if conditions have been met for HAMMER PULSE to be output low. If conditions have been met, then HAMMER PULSE will be output low immediately; if conditions have not been met, the BEQ HP0 instruction branches around the instruction sequence that outputs HAMMER PULSE low:

```
        LDA  A    $C000  ⎫
        AND  A    #$40   ⎬   Test hammer pulse output conditions
        BEQ       HP0    ⎭
        LDA  A    $C001  ⎫
        AND  A    #$FB   ⎬   If conditions have been met, output HAMMER PULSE low
        STA  A    $C001  ⎭
HP0 ──► LDX       #DELY
```
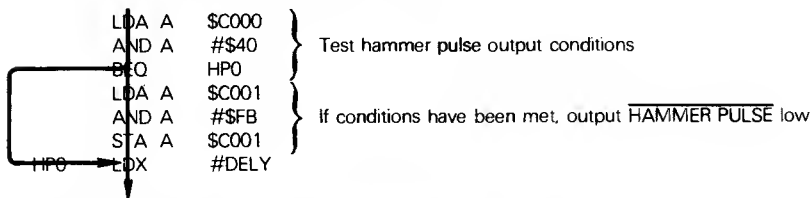
We output HAMMER PULSE low before starting to compute   ┌──────────────┐
the duration of the time pulse; why is this? The reason is to save   │ **EVENT** │
time. Instructions which compute the length of the time delay can be ex-   │ **SEQUENCE** │
ecuted at the beginning of the time delay: └──────────────┘

STA instruction execution

Set HAMMER PULSE low
Execute instructions which compute time delay length
Duration of computed time delay
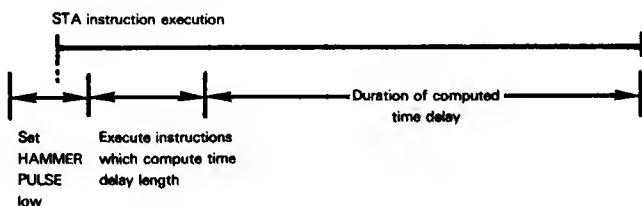
We could just as easily have computed the time delay, then set HAMMER PULSE low; then executed the time delay; events would have occurred chronologically as follows:

STA instruction execution

Compute length of time delay
Set HAMMER PULSE low
Total time delay
Duration of computed time delay
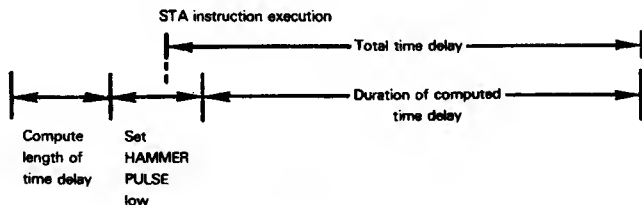
Overlapping events in time makes a lot more sense.

The actual method used to compute time the time delay needs a little explanation. **At the end of our program, there will be 12 bytes of memory in which six 16-bit constants are stored.** This is how the source program will look:

```
          BNE       TDLY
OUTPUT HAMMER PULSE HIGH AGAIN
          LDA  A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
          ORA  A    #4       SET BIT 2 TO 1
          STA  A    $C001    OUTPUT RESULT
          -
          -
          -
          -
          -
          -
          ORG       DELY + 2
          FDB       pppp     H1 TIME DELAY
          FDB       qqqq     H2 TIME DELAY
          FDB       rrrr     H3 TIME DELAY
          FDB       ssss     H4 TIME DELAY
          FDB       tttt     H5 TIME DELAY
          FDB       uuuu     H6 TIME DELAY
```

The letters p, q, r, s, t and u have been used to represent hexadecimal values. The six time delays can be represented by any numeric values, ranging from 0000_{16} through $FFFF_{16}$.

The address of the first memory byte in which the first time delay is stored is given by the expression DELY+2. Suppose this memory location happened to be 2138:

Arbitrary Memory Address	DATA MEMORY
2138	p p
2139	p p
213A	q q
213B	q q
213C	r r
213D	r r
213E	s s
213F	s s
2140	t t
2141	t t
	etc.
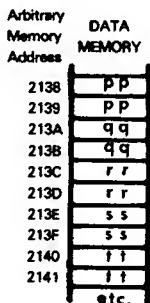
DELY is a label to which the value 2136 must be assigned. This assignment is made using an Equate directive, which would appear at the beginning of the program as follows:

```
          DELY      EQU      $2136
```

Now we begin our computation of the time delay by loading the address DELY into the Index register. Assume that the label DELY has the value 2136, as illustrated above. After the LDX #DELY instruction has been executed. this is the situation:
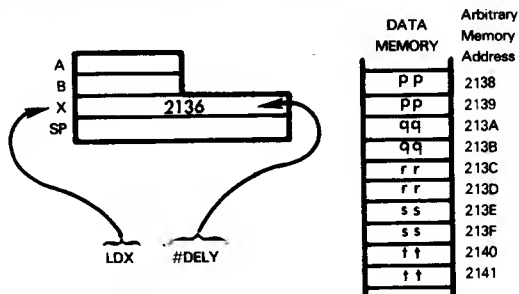
The next instruction, LDA A H1H6, loads the contents of I/O Port C into Accumulator A. The memory address which causes the I/O port to select itself is represented by the label H1H6. This memory address is $E000_{16}$; thus H1H6 would have to be assigned the value $E000_{16}$ using an Equate directive at the beginning of the program, as follows:

```
DELY     EQU     $2136
H1H6     EQU     $E000
```

From our discussion of input signals, recall that of the six inputs H1 through H6, one signal will be high while the other five signals are low. Therefore, **after the LDA instruction has executed, Accumulator A will contain a 1 in one of the six low order bits:**
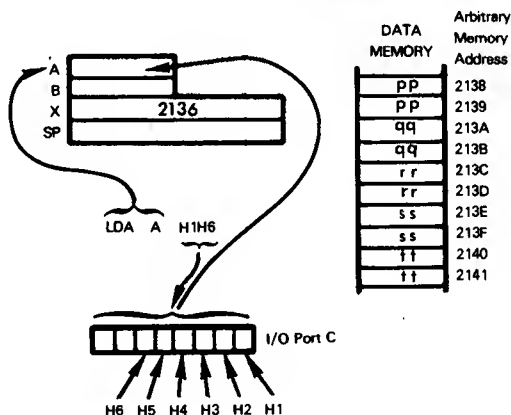
We can compute the address of the required time delay by adding 2 to the contents of the Index register a number of times given by the position of the Accumulator A 1 bit. This may be illustrated as follows:

DATA MEMORY ADDRESS COMPUTATION

① Shift Accumulator A contents right one bit, and into Carry:
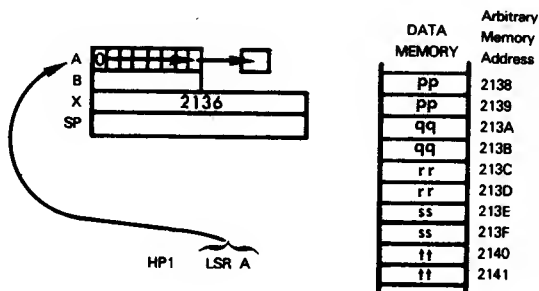
② Add 2 to the Index register:
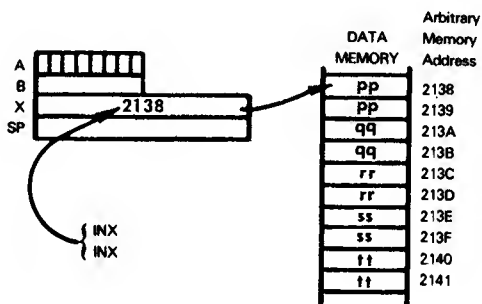
③ If Carry status is not 1, go back to ① ; otherwise Index register contains the correct address.

The logic to make the required address addition is provided by these four instructions:

HP1	LSR A	SHIFT ACCUMULATOR RIGHT WITH CARRY
	INX	INCREMENT INDEX REGISTER BY 2
	INX	
	BCC HP1	IF CARRY IS CLEAR, SHIFT AND INCREMENT AGAIN

When the BCC instruction causes program execution to continue with the next sequential instruction, rather than branching back to HP1, the Index register will contain the address of the initial time delay constant's first byte. This constant must be loaded back into the Index register, since we are going to use the long time delay instruction sequence; this instruction sequence

3-66

decrements the Index register contents till it reaches a 0 value. The LDX 0,X instruction loads the Index register using direct, indexed addressing. This may be illustrated as follows:
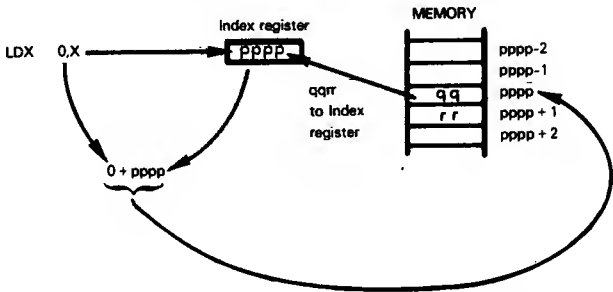
Thus the Index register has been loaded with the contents of the memory location it was just addressing (qq), plus the contents of the next memory byte (rr); the Index register now contains the correct initial constant for a long time delay instruction loop.

The actual time delay is created by this instruction loop, which was described in Chapter 2:

```
TDLY    DEX
        BNE     TDLY
```

The last three instructions output HAMMER PULSE high, without making any test for whether HAMMER PULSE was low. This logic will work since outputing HAMMER PULSE high, if it was already high, will have no discernible effect. Under these circumstances, the time required to execute the last three instructions is simply wasted. Since it would take three instructions to test if HAMMER PULSE had been set low, the waste is justified.

Let us now give a little thought to the time it will take to compute the time delay. Execution times for relevant instructions are listed as follows:

TIME DELAY COMPUTATION

Cycles		Instruction		
4		LDA A	$C001	
2		AND A	#$FB	
5		STA A	$C001	◄— HAMMER PULSE low starts here
3	HP0	LDX	#DELY	
4		LDA A	H1H6	
2	HP1	LSR A		These four instructions will be executed
4		INX		between 1 and 6 times. 14 cycles are in
4		INX		the loop
4		BCC	HP1	
6		LDX	0,X	
4	TDLY	DEX		These two instructions constitute the
4		BNE	TDLY	time delay. 8 cycles are in this loop
4		LDA A	$C001	
2		ORA A	#4	
5		STA A	$C001	◄— HAMMER PULSE low ends here
57				

3-67

Assuming a 1 microsecond clock, the time taken to initiate and terminate the HAMMER PULSE signal is given by:

57 - 8 - 14 + 14N microseconds

where N is a number between 1 and 6, representing the bit position of Accumulator A that is set to 1. **Thus initiation and termination time will vary between 49 microseconds and 119 microseconds.** The shortest time applies to N=1 (H1) whereas the longest time applies to N=6 (H6).

These times must be subtracted from the delays subsequently generated. For example, suppose H1 high requires the 555 to output a one-shot signal which is high for 1.65 milliseconds (approximately); then a delay of 1.6 milliseconds, added to a set up time of 49 microseconds will suffice.

THE PW RELEASE ENABLE FLIP-FLOP

As soon as the 555 one-shot output becomes low again, flip-flop FFC is simulated switching off. When FFC switches off, its Q̄ output makes a low-to-high transition and this triggers the PW RELEASE ENABLE one-shot. This is a 74121 one-shot, identified by the 36 at approximately co-ordinate E2. The purpose of this one-shot is to allow the printhammer time to settle back before any attempt is made to reposition the printwheel. **This was illustrated as the fixed, hammer return and settling time delay.**

SIMULATING THE PW RELEASE ENABLE FLIP-FLOP

This is really a two-part simulation; first we must simulate flip-flop FFC switching off, then we must execute an appropriate time delay. A 3 millisecond time delay is sufficient. Instructions which turn flip-flop FFC off will execute within the 3 millisecond time delay. The computed time delay will therefore be a little less than 3 milliseconds. Here is the appropriate instruction sequence:

```
OUTPUT HAMMER PULSE HIGH AGAIN
        LDA  A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
        ORA  A    #4       SET BIT 2 TO 1
        STA  A    $C001    OUTPUT RESULT
SWITCH FLIP-FLOP FFC OFF
        AND  B    #$FB     SET BIT 2 TO 0
        STA  B    $E001
EXECUTE A 3 MILLISECOND TIME DELAY
        LDX       #374     LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR1    DEX                DECREMENT INDEX REGISTER
        BNE       PWR1     REDECREMENT IF NOT ZERO
```

Notice that the initial time constant has been identified as a decimal number, 374. The time constant could be specified as a hexadecimal number thus:

        LDX   #$176

The three instructions which precede the time delay loop (AND, STA and LDX) execute in 10 microseconds, and the two instructions in the delay loop execute in 8 microseconds. Therefore the total delay time is given by the equation:

374 x 8 + 10 = 3002 microseconds

To be honest, the 3 millisecond time delay is not a critical number; 2.5 or 3.5 milliseconds would probably do just as well, so our worrying about 10 microseconds is not meaningful in this instance. Nevertheless, in your next application, the duration of a time delay may be very critical; then the timing considerations discussed above will be very meaningful.

In order to determine what happens at the conclusion of the PW RELEASE time delay, we must look at the FFC Q and Q̄ outputs. The Q output connects to the START RIBBON MOTION PULSE AND gate, and to the 555 one-shot trigger logic; in neither case does the O high-to-low transition have any effect. The START RIBBON MOTION pulse signal is already low and the 555 one-shot is triggered by a high-to-low Q transition. The low-to-high transition simply raises the trigger signal to a high level which requires no simulation:
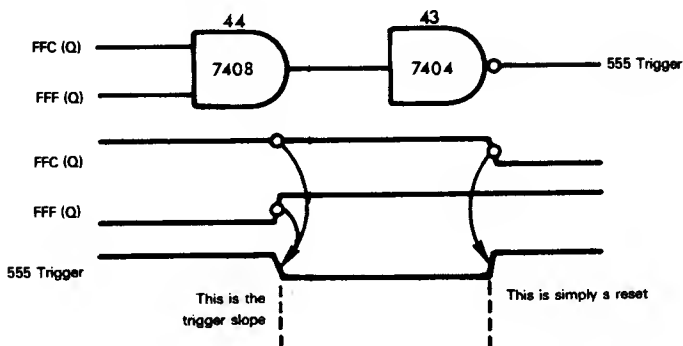
The FFC (Q̄) output is ANDed with the PW RELEASE ENABLE Q̄ one-shot in order to generate the FFD (K) input. The FFD (J) input comes directly from FFC (Q), therefore **as soon as the PW RELEASE ENABLE one-shot goes high again, FFD will receive a low J input and a high K input:**
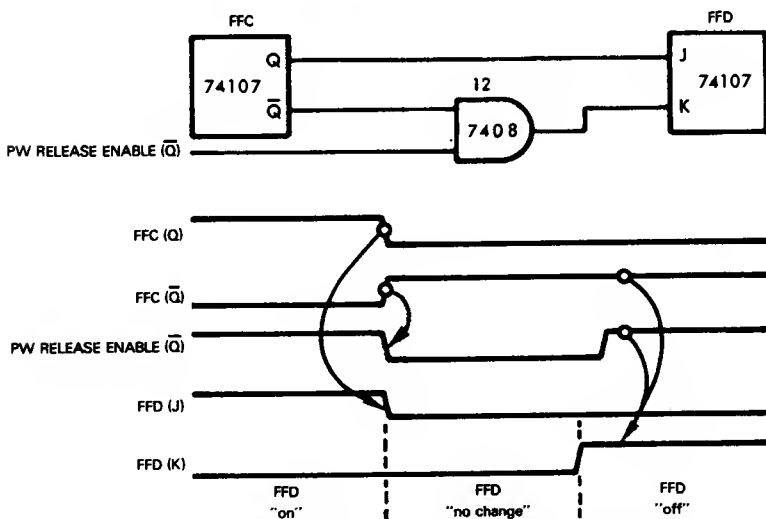
A low J and high K input to flip-flop FFD switches this flip-flop off; and that triggers the PW READY ENABLE one-shot.

SIMULATING THE PW READY ENABLE ONE-SHOT

Logic associated with this one-shot is almost identical to the PW RELEASE ENABLE one-shot. FFD switching off causes a low-to-high $\overline{Q}$ output, which triggers the PW READY ENABLE one-shot.

We must now simulate a 2 millisecond time delay; otherwise the next instruction sequence is almost identical to the PW RELEASE ENABLE one-shot simulation and may be illustrated as follows:

```
EXECUTE A 3 MILLISECOND TIME DELAY
        LDX     #374        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR1    DEX                 DECREMENT INDEX REGISTER
        BNE     PWR1        REDECREMENT IF NOT ZERO
SWITCH FLIP-FLOP FFD OFF
        AND B   #$F7        SET BIT 3 TO 0
        STA B   $E001       OUTPUT RESULT
EXECUTE A 2 MILLISECOND TIME DELAY
        LDX     #249        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR2    DEX                 DECREMENT INDEX REGISTER
        BNE     PWR2        REDECREMENT IF NOT ZERO
```

When FFD switches off, the PW REL output goes high again. Here is the PW REL creation logic:
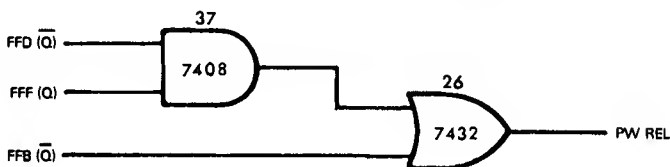
FFB ($\overline{Q}$) is still low at this time. But FFD ($\overline{Q}$) and FFF (Q) are both high so AND gate 37 outputs a high level which passes through OR gate 26 to set PW REL high.

These instructions set PW REL high:

```
EXECUTE A 2 MILLISECOND TIME DELAY
        LDX     #249        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR2    DEX                 DECREMENT INDEX REGISTER
        BNE     PWR2        REDECREMENT IF NOT ZERO
SET PW REL HIGH
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #1          SET BIT 0 TO 1
        STA A   $C001
```

Now the whole print cycle ends in a hurry. The flip-flop FFD Q and $\overline{Q}$ outputs become the FFE J and K inputs. Q is first ANDed with FFI which, at this time, is constantly high; therefore the moment FFD switches off, FFE receives a low J input.

The FFE (K) input does not go high until the end of the PW READY ENABLE one-shot, since the PW READY ENABLE $\overline{Q}$ output is ANDed with $\overline{Q}$ from FFD in order to generate FFE (K).

FFE switching off is our next chronological event.

FFE switching off, in turn, causes FFB and FFF to switch off. FFB is switched off by the low-to-high transition of FFE ($\overline{Q}$) which becomes the FFB clock input. FFF switches off because its J and K inputs are tied directly to the Q and $\overline{Q}$ outputs of FFE.

Once FFB and FFF have switched off, all conditions have been met for CH RDY to go high again, providing $\overline{\text{EOR DET}}$ is not signaling the end of ribbon:


```
EXECUTE A 2 MILLISECOND TIME DELAY
        LDX     #249        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR2    DEX                 DECREMENT INDEX REGISTER
        BNE     PWR2        REDECREMENT IF NOT ZERO
SET PW REL HIGH
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #1          SET BIT 0 TO 1
        STA A   $C001
TURN OFF FLIP-FLOPS FFB, FFE AND FFF
        AND B   #$AF        RESET BITS 4 AND 6 TO 0
        ORA B   #$22        SET BITS 5 AND 1 TO 1
        STA B   $E001       OUTPUT RESULT
SET CH RDY HIGH
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #2          SET BIT 1 TO 1
        STA A   $C001       OUTPUT RESULT
BRANCH TO TEST FOR VALID END OF PRINT CYCLE
        JMP     LOP1
```

SIMULATION SUMMARY

The complete simulation program developed in this chapter is given in Figure 3-3.

We can conclude that an absolutely exact, one-for-one simulation of digital logic using assembly language instructions within a microcomputer system is not feasible; but then it is not particularly desirable.

If you are not a digital logic designer, you will probably be very confused by the various signal combinations required within the logic of Figure 3-1. A great deal of what is going on has nothing to do with the ultimate requirements of the Qume printer; rather, it reflects one logic designer's internal logic implementation, aimed at insuring appropriate external signal sequences under all conceivable circumstances.

If you are a logic designer, chances are you would have implemented the specific requirements of the Qume printer interface in a totally different way; you may even be grumbling at this implementation.

The important point to bear in mind is that digital logic contains innumerable subtleties which are specific to discrete logic devices. These subtleties are not tied to the requirements of the overall implementation.

Now assembly language has its own set of subtleties, which also have nothing to do with the ultimate implementation; rather, they are aimed at making most effective use of individual instructions or instruction sequences.

It should therefore come as no surprise that an exact duplication of digital logic, using assembly language, is neither feasible nor desirable. So we will move away from digital logic and start treating a problem from a programming viewpoint.

The principal difference between digital logic and assembly language is that assembly language treats events chronologically, while digital logic segregates logic into functional nodes. Thus, one logic device may be responsible for a number of events occurring at different times during any logic cycle; when translated into an assembly language program, each event becomes an isolated instruction sequence.

ASSEMBLY
LANGUAGE
VERSUS
DIGITAL
LOGIC

In Figure 3-1 for example, the print cycle began with a cascade of flip-flops switching on and ended with the same flip-flops switching off. In many cases a flip-flop switching on triggered one event, while the same flip-flop switching off triggered an entirely different event. Within an assembly language program, the two events will have nothing in common. Each event will be represented by a completely independent instruction sequence occurring at substantially different parts of the program.

The other major difference between digital logic and assembly language is the concept of timing. Within synchronous digital logic, as illustrated in Figure 3-1, timing is bound to clock signals and the need for clean signal interactions. Within an assembly language program, timing results strictly from the sequence in which instructions are executed. Moreover, whereas components in a digital logic circuit may switch and operate in parallel; within an assembly language program everything must occur serially.

Now the key concept to grasp from this chapter is that there is nothing innately correct about digital logic as a means of implementing anything. The fact that we have been unable to exactly duplicate digital logic using assembly language instructions does not mean that assembly language is in any way inferior; it simply means that assembly language is going to do the job in a different way.

Having spent our time in Chapter 3 drawing direct parallels between assembly language and digital logic, we will now abandon any attempt to favor digital logic. Moving on to Chapter 4, the logic illustrated in Figure 3-1 will be resimulated — but from the programmer's point of view.

```
TEST FOR VALID END OF PRINT CYCLE
LOP1    LDA  A    $C001    INPUT I/O PORT B CONTENTS TO ACCUMULATOR A
        ROL  A             SHIFT BIT 7 INTO CARRY
        BCC       LOP1     IF ZERO IN CARRY, STAY IN PRINT CYCLE
IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
INITIALLY SET I/O PORT D BITS 6, 4, 3 AND 2 TO 0, BITS 5, 1 AND 0 TO 1
        LDA  B    $E001    INPUT I/O PORT D TO ACCUMULATOR B
        ORA  B    #$23     SET BITS 5, 1 AND 0 TO 1
        AND  B    #$A3     RESET BITS 6, 4, 3 AND 2 TO 0
        STA  B    $E001    RETURN RESULT
TEST FOR RETURN STROBE LOW
L10     LDA  A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$10     ISOLATE RETURN STROBE
        BEQ       FFB      IF IT IS 0, JUMP TO FFB SIMULATION
SIMULATION OF FFA AND ASSOCIATED LOGIC
LOAD I/O PORT B CONTENTS INTO ACCUMULATOR A AND ISOLATE BITS 1, 5 AND
6 FOR CH RDY, PW STROBE AND RESET, RESPECTIVELY
        LDA  A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
        AND  A    #$62     ISOLATE BITS 6, 5 AND 1
        CMP  A    #$22     IF RESET=0, CH RDY=1 AND PW STROBE=1, START NEW
                           PRINT CYCLE
        BNE       L10      OTHERWISE RETURN TO L10 TO START A NEW
                           PRINT CYCLE
        AND  B    #$FE     SET I/O PORT D BIT 0 TO 0
        STA  B    $E001
```

Figure 3-5. The Complete Simulation Program

NEW PRINT CYCLE SEQUENCE STARTS HERE
SIMULATE FLIP-FLOP FFB SWITCHING ON

```
FFB     AND B    #$FD       RESET BIT 1 TO 0
        STA B    $E001      RESTORE RESULT
```

SIMULATE 7411 AND GATE SWITCHING CH RDY LOW. ALSO 7432 OR GATE
SWITCHES PW REL LOW

```
        LDA A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND A    #$FC       RESET BITS 0 AND 1 TO 0
        STA A    $C001      RESTORE RESULT
```

CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT D TO 1
ALSO SIMULATE FFC AND FFD TURNING ON. SET BIT 2 OF I/O PORT D TO 1

```
        ORA B    #$0D       SET BITS 3, 2 AND 0 TO 1
        STA B    $E001      RESTORE RESULT
```

PULSE START RIBBON MOTION HIGH

```
        ORA A    #8         SET BIT 3 HIGH
        STA A    $C001      OUTPUT TO I/O PORT B
        AND A    #$F7       SET BIT 3 LOW
        STA A    $C001      OUTPUT TO I/O PORT B
```

TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY

```
VLDC    LDA A    $C000      INPUT I/O PORT A TO ACCUMULATOR A
        ROL A               SHIFT BIT 7 INTO CARRY
        BCC      VLDC       STAY IN LOOP IF CARRY IS ZERO
```

AT END OF DELAY SIMULATE FFE SWITCHING ON

```
        AND B    #$DF       RESET BIT 5
        ORA B    #$10       SET BIT 4
        STA B    $E001      OUTPUT THE RESULT
```

SIMULATE 2 MS PW SETTLING TIME DELAY

```
        LDX      #$FA       LOAD INITIAL TIME DELAY CONSTANT
PWS     DEX                 DECREMENT INDEX REGISTER
        BNE      PWS        REDECREMENT IF NOT ZERO
```

SIMULATE FLIP-FLOP FFF SWITCHING ON

```
FFF     LDA A    $C000      INPUT I/O PORT A CONTENTS TO ACCUMULATOR A
        COM A               COMPLEMENT TO TEST FOR 1 BITS
        AND A    #$1F       ISOLATE BITS 0 THROUGH 4
        BNE      FFF        IF ANY BITS ARE 1, STAY IN LOOP
        ORA B    #$40       SET BIT 6 OF I/O PORT D TO 1
        STA B    $E001
```

TEST HAMMER ENABLE FF

```
        LDA A    $C000      INPUT I/O PORT A TO ACCUMULATOR A
        AND A    #$40       ISOLATE BIT 6
        BEO      HP0        IF ZERO, BYPASS SETTING HAMMER PULSE LOW
```

HAMMER ENABLE FF IS HIGH, SO HAMMER PULSE MUST BE OUTPUT LOW.
THEREFORE SET BIT 2 OF I/O PORT B TO 0

```
        LDA A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND A    #$FB       SET BIT 2 TO 0
        STA A    $C001      OUTPUT RESULT
```

Figure 3-5. The Complete Simulation Program (Continued)

```
COMPUTE TIME DELAY
HP0       LDX       #DELY       LOAD DELAY BASE ADDRESS INTO INDEX REGISTER
          LDA  A    H1H6        LOAD SELECTOR INTO ACCUMULATOR A
HP1       LSR  A                SHIFT ACCUMULATOR RIGHT WITH CARRY
          INX                   INCREMENT INDEX REGISTER BY 2
          INX
          BCC       HP1         IF CARRY IS CLEAR SHIFT AND INCREMENT AGAIN
          LDX       0,X         LOAD 16-BIT DELAY COUNTER INTO INDEX REGISTER
TDLY      DEX                   EXECUTE TIME DELAY LOOP
          BNE       TDLY
OUTPUT HAMMER PULSE HIGH AGAIN
          LDA  A    $C001       INPUT I/O PORT B TO ACCUMULATOR A
          ORA  A    #4          SET BIT 2 TO 1
          STA  A    $C001       OUTPUT RESULT
SWITCH FLIP-FLOP FFC OFF
          AND  B    #$FB        SET BIT 2 TO 0
          STA  B    $E001
EXECUTE A 3 MILLISECOND TIME DELAY
          LDX       #374        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR1      DEX                   DECREMENT INDEX REGISTER
          BNE       PWR1        REDECREMENT IF NOT ZERO
SWITCH FLIP-FLOP FFD OFF
          AND  B    #$F7        SET BIT 3 TO 0
          STA  B    $E001       OUTPUT RESULT
EXECUTE A 2 MILLISECOND TIME DELAY
          LDX       #249        LOAD INITIAL TIME CONSTANT INTO INDEX REGISTER
PWR2      DEX                   DECREMENT INDEX REGISTER
          BNE       PWR2        REDECREMENT IF NOT ZERO
SET PW REL HIGH
          LDA  A    $C001       INPUT I/O PORT B TO ACCUMULATOR A
          ORA  A    #1          SET BIT 0 TO 1
          STA  A    $C001
TURN OFF FLIP-FLOPS FFB, FFE AND FFF
          AND  B    #$AF        RESET BITS 4 AND 6 TO 0
          ORA  B    #$22        SET BITS 5 AND 1 TO 1
          STA  B    $E001       OUTPUT RESULT
SET CH RDY HIGH
          LDA  A    $C001       INPUT I/O PORT B TO ACCUMULATOR A
          ORA  A    #2          SET BIT 1 TO 1
          STA  A    $C001       OUTPUT RESULT
BRANCH TO TEST FOR VALID END OF PRINT CYCLE
          JMP       LOP1
DELAY COUNT TABLE
          ORG       DELY + 2
          FDB       PPPP        H1 TIME DELAY
          FDB       QQQQ        H2 TIME DELAY
          FDB       RRRR        H3 TIME DELAY
          FDB       SSSS        H4 TIME DELAY
          FDB       TTTT        H5 TIME DELAY
          FDB       UUUU        H6 TIME DELAY
```

The letters P, Q, R, S, T and U represent hexadecimal digits.

Figure 3-5. The Complete Simulation Program (Continued)

Chapter 4
A SIMPLE PROGRAM

The problems sssociated with simulating digital logic, es we did in Chapter 3, can be ettributed to one fect: we tried to divide logic into a number of isoleted transfer functions, eech of which corresponded to e digitel logic device. We are now going to abandon digital and combinatorial logic. pretend it does not exist and take another look at Figures 3-1 and 3-2.

ASSEMBLY LANGUAGE TIMING VERSUS DIGITAL LOGIC TIMING

Returning to Figure 3-1, simply ignore everything thet exists between the left snd right hand margins of the figure. What remains is a set of input signals end s set of output signals. The

TRANSFER FUNCTION

output signels are related to the input signels by a set of transfer functions which have nothing to do with digital logic devices.

The transfer functions for Figure 3-1 are loosely represented by the timing diegrem in Figure 3-2. Whet does "loosely represented" meen? It means that timing which relates to system requirements is mixed indiscriminetely with timing that simply reflects the needs of digital logic. We can ebandon timing consideretions thet simply reflect the needs of digital logic. To be specific, the printhammer must still be fired by outputting one of six solenoid pulses; the various movement and settling delays must elso be mainteined. But we can ebandon time delays that separate one signel's change of state from another simply to keep the digitel logic clean.

From the progremmer's point of view, therefore, the timing diagram illustrated in Figure 4-1 is a perfectly valid substitution for the logic designer's timing diagram illustrated in Figure 3-2.

INPUT AND OUTPUT SIGNALS

Looking et Figure 4-1 you will see thst we have absndoned e lot more than minor timing deleys; we have also ebandoned most of our signels. But there is a simple criterion for determining whether a signsl is really necessary within a microcomputer system. This is the criterion: if the signal is uniquely associated with reel time events in logic external to the microcomputer system, then the signal must remain. If the source and destination of the signal are within the microcomputer system "blsck box", then the signal may be abandoned. Based on this criterion, let us take another look at our input and output signals.

First consider the input signels.

RETURN STROBE and PW STROBE are meaningless signals. As digital logic, these two signals are print cycle sequence initiators. Within

INPUT SIGNALS

an assembly language program, jumping to the first instruction of a sequence is all the initiation you need. The fact that RETURN STROBE represents a print cycle during which the printhammer is not fired is unimportant, because HAMMER ENABLE is used to actually suppress HAMMER PULSE.

We will combine the various hammer firing inhibit signals into one hammer status input. There are five such signals: PFL REL, RIB LIFT RDY, RIBBON ADVANCE, PFR REL and CA REL. Each of these signals owes its origin to different logic external to Figure 3-1; in the digital logic implementation, these signals are ANDed in order to create a master HAMMER INTERLOCK signal. In our assembly language implementation we will wire-OR all of these external signals to a single pin which becomes a HAMMER INTERLOCK status.

RESET will be maintained as a master RESET signal tied to the CPU RESET pin. RESET can therefore be ignored by the assembly language program; however, recall that once RESET is activated, program execution is going to resume with the instruction stored at the memory location whose address is fetched from memory bytes $FFFF_{16}$ and $FFFE_{16}$.

EOR DET will be maintained. This is the signal which detects end of ribbon and prevents a print cycle from ever ending, thus inhibiting further character printing after the ribbon is exhausted.

HAMMER ENABLE FF must be maintained; it suppresses the printhammer firing pulse during printwheel repositioning print cycles.

The function performed by the six hammer pulse length signals, H1 through H6, must remain, but the signals themselves will disappear. Instead of using six pins of an I/O port to identify hammer pulse width, we are going to create time delays directly from ASCII character codes.

Let us now turn our attention to the output signals.

To begin with, we can eliminate all of the flip-flop outputs. The boundary of each time interval within the print cycle is already identified by an existing signal changing state. If more than one external logic event must be triggered by a transition from one time interval to the next, there is nothing to stop the appropriate signal from being buffered externally, then used to trigger numerous external logic events. Within the microcomputer program, there is no reason why duplicate signals should be output simply to identify the transition from one print cycle time interval to the next.

The remaining output signals are maintained. It is possible that some of these signals would disappear if additional external logic were replaced by more assembly language programs within the microcomputer system; but given the bounds of the problem, as stated, the remaining signals are needed in order to define the print cycle time intervals.
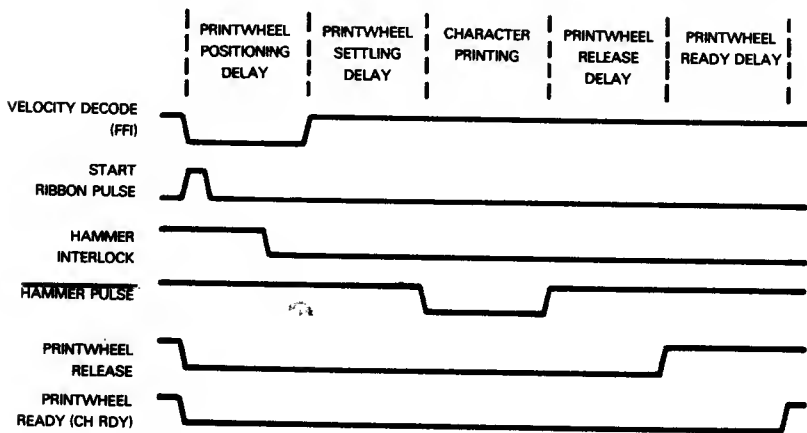
Figure 4-1. Timing For Figure 3-1, From The Programmer's Viewpoint

Given our new, simplified set of signals, we can eliminate one MC6820 PIA; for the single remaining MC6820 PIA, I/O ports and pins are assigned as follows:
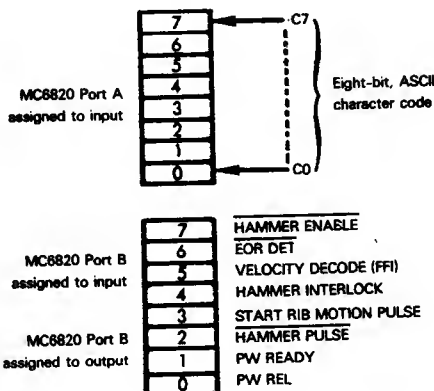
MICROCOMPUTER DEVICE CONFIGURATION

We are now in a position to select the devices needed for program implementation. The selection is really quite straightforward; in addition to the CPU, we will need one MC6820 Peripheral Interface Adapter, some read-only memory for program storage and some read/write memory for general data storage. The CPU, in reality, consists of two devices: the CPU itself and a Clock chip. Combining these devices, Figure 4-2 illustrates the microcomputer system which results. Now if you don't immediately understand Figure 4-2 do not despair, there are only a few aspects of this figure which are consequential to our immediate discussion.

GENERAL DESIGN CONCEPTS

This is the most important concept to derive from Figure 4-2: when designing logic by writing assembly language programs within a microcomputer system, the program you write is going to be highly dependent upon the device configuration. There is nothing unique about the way in which devices have been combined as illustrated in Figure 4-2; alternative configurations would be equally viable. The assembly language programs created, however, might differ markedly from one microcomputer configuration to the next and this is a factor you should not lose sight of when writing microcomputer programs. Also, do not be afraid of modifying the selected hardware configuration; that is precisely what we will do in Chapter 5. Microcomputer device configuration and assembly language programming interact strongly and should not be separated. These two steps should be within one iterative loop. During the early stages of writing a microcomputer program, you should assume that in the course of writing the assembly language program, you will discover features of the hardware that can be improved; that in turn means the program will have to be rewritten.

This is a good point at which to bring up one of the reasons why higher level languages are not desirable when you are programming a microcomputer to replace digital logic. Higher level languages are problem-oriented. For example, it is hard to look at a PL/M program statement and visualize the exact way in which data will be moved around a microcomputer system in response to the statement's execution. It is even harder to relate PL/M programs to exact device configurations. Assembly language, on the other hand, has a one-for-one relationship with your hardware.

MC6870A TWO-PHASE CLOCK

You can use a variety of different clock devices with an MC6800 microprocessor.
The principal difference between one device and the next is the number of functions in addition
to the simple clock signals which the clock device provides. All of these clock devices are de-
scribed in detail in Chapter 6 of "An Introduction To Microcomputers: Volume II — Some Real
Products".

Figure 4-2. MC6800 Microcomputer Configuration

Figure 4-2 represents the simplest implementation of our microcomputer system and its needs are met by the MC6870A two-phase clock device. This clock device has an on-chip oscillator so that no external crystal is required. In addition to developing the $\Phi 1$ and $\Phi 2$ clock signals required by the MC6800 microprocessor, the MC6870A two-phase clock develops a TTL level $\Phi 2$ clock. This additional $\Phi 2$ clock is needed in order to synchronize logic within the MC6820 Peripheral Interface Adapter. The E input to the Peripheral Interface Adapter is connected directly to the $\Phi 2$ (TTL) signal.

The MC6870A two-phase clock provides no other logic. For small MC6800 microcomputer systems, the MC6870A two-phase clock is the device of preference; since it requires no external crystal, it is simple to use.

MC6820 PERIPHERAL INTERFACE ADAPTER (PIA)

Now let us turn our attention to the specific way in which devices have been incorporated into Figure 4-2.

The MC6820 Peripheral Interface Adapter will respond to memory addresses as follows:
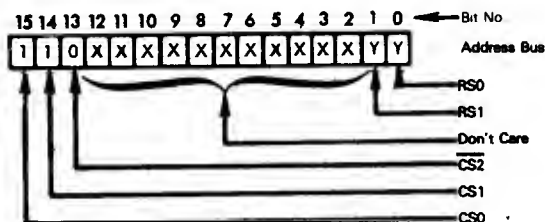
We will assume that all of the don't care address bits are 0; as a result **we will use the four addresses** $C000_{16}$, $C001_{16}$, $C002_{16}$, **and $C003_{16}$ to address the single MC6820 PIA in Figure 4-2. The four addresees will access PIA locations as** follows:

CHIP SELECT IN SIMPLE SYSTEMS

$C000_{16}$: I/O Port A or Data Direction Register A

$C001_{16}$: I/O Port B or Data Direction Register 8

$C002_{16}$: Control Register A

$C003_{16}$: Control Register B

If a microcomputer configuration contains a large number of Peripheral Interface Adapters the chip select logic may become a little more complex. If a PIA is to respond to four unique memory addresses, excluding all others, then the chip select input must be created by combining all 16 address lines in some unique way.

Suppose the MC6820 PIA in Figure 4-2 must respond to memory addresses $C000_{16}$, $C001_{16}$, $C002_{16}$ **and $C003_{16}$ only.** Now all of the don't care signal lines must input to logic which

CHIP SELECT IN LARGER SYSTEMS

is true only when these signal lines are all low. **This is one way of creating chip select logic:**
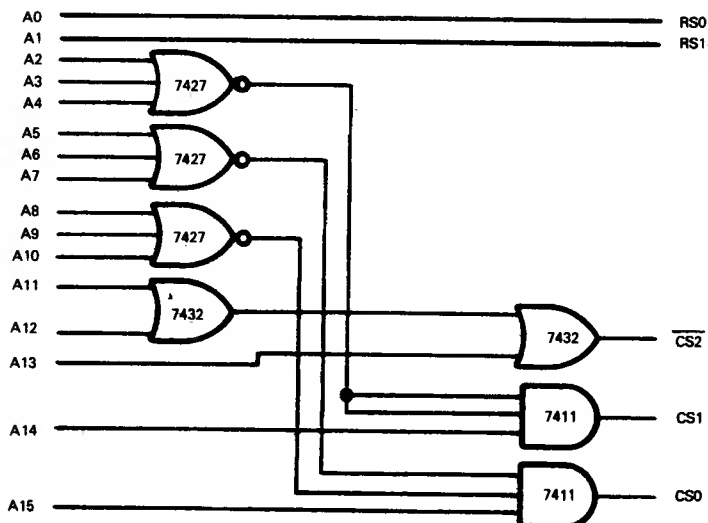
The CS1 and CS0 signals can be created using a 7427 Triple 3-Input Positive-NOR gate and two of the three gates in a 7411 Triple 3-Input Positive-AND gate. The CS2 signal comes from two of the four gates in a 7432 Quadruple 2-Input Positive-OR gate.

Given the above select logic, the MC6820 will consider itself selected if and only if one of the four specified addresses is output on the Address Bus.

The data direction and port utilization illustrated for the MC6820 PIA in Figure 4-2 is not a hardware feature. At any time port utilization may be modified by writing the appropriate control word into the Data Direction registers and Control register of the MC6820.

The RESET logic needs comment. Instead of testing for a Reset condition in between print cycles, as we did in Chapter 3, **we are going to use a hardware RESET signal,** but in a microcomputer environment.

| RESET |
| LOGIC |

The RESET signal connected to the MC6820 PIA will clear all registers in the PIA. This will result in I/O ports being defined as inputs; control options resulting from all 0s in Control register bits will be in effect. At some point following a RESET we must execute instructions which load Data Direction registers and Control registers appropriately.

| MC6820 |
| RESET |
| LOGIC |

The RESET signal being input to the MC6800 microprocessor causes the microprocessor to reinitialize itself by loading into the Program Counter a 16-bit address which is stored in memory locations $FFFE_{16}$ and $FFFF_{16}$. This logic is described in Chapter 6 of "An Introduction To Microcomputers: Volume II — Some Real Products".

Memory select logic illustrated in Figure 4-2 will satisfy RESET logic requirements.

ROM AND RAM MEMORY

An MCM6830A provides our microcomputer system with 1024 bytes of read-only memory. Four select lines, plus ten address lines create ROM addresses as follows:

ROM ADDRESSES
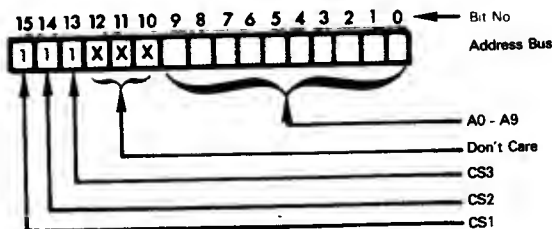
If the don't care bits are assumed equal to 0, then the ROM device will be selected by addresses in the range $E000_{16}$ through $E3FF_{16}$. If the don't care bits are assumed equal to 1 then the ROM device will be selected by addresses in the range $FC00_{16}$ through $FFFF_{16}$.

As we have just seen, following a Reset, the MC6800 microprocessor will fetch its initialization address from memory locations $FFFE_{16}$ and $FFFF_{16}$. These will represent the two highest memory locations of the ROM device as implemented in Figure 4-2, providing the don't care bits are assumed equal to 1.

Notice that under no circumstance will the ROM address space conflict with the MC6820 PIA; address line A13 equal to 0 is a prerequisite for the PIA to be selected, while address line A13 must be 1 for the ROM device to be selected.

The CS0 chip select of the MCM6830A device is connected to the R/$\overline{W}$ control. Thus, in order to complete ROM device selection, a read operation must be specified.

MEMORY DEVICE SELECT USING R/$\overline{W}$ CONTROL

Were we to assign a unique address space to the 1024 ROM bytes, then the three don't care bits would have to have some specific value which contributes to the device select logic. Since addresses $FFFE_{16}$ and $FFFF_{16}$ are required by RESET logic, our unique address space must be based on the three don't care bits all having values of 1. For example, a four input AND gate could be used to generate CS3 as follows:

Now the 1K bytes of ROM memory would be selected only by addresses in the range $FC00_{16}$ through $FFFF_{16}$.

An MCM6810A device provides our microcomputer system with 128 bytes of read/write memory. This memory device has six chip select pins which cause the RAM device to be selected by addresses in the range 0000_{16} through $007F_{16}$. We have selected the first 128 addresses for our read/write memory, since this is common practice in microcomputer systems. In fact, the MC6800 instruction set has direct addressing instructions which are two bytes long, rather than three bytes long, assuming that read/write memory occupies the first 256 bytes of memory. It is for this reason that the two chip select lines which expect high inputs receive inverted address data.

RAM MEMORY ADDRESS

In summary, addresses for the microcomputer system illustrated in Figure 4-2 will be interpreted as follows:

MEMORY ADDRESSES	SELECT
0000_{16} - $007F_{16}$	Read/Write memory
$C000_{16}$ - $C003_{16}$	MC6820 registers
$FC00_{16}$ - $FFFF_{16}$	Read-only memory

SYSTEM INITIALIZATION

Let us now turn our attention to system operations.

When the system is initialized, "in between print cycles" conditions must be re-established immediately. These are the necessary steps:

1) If the printhammer has been fired, discontinue the firing pulse and allow the printhammer time to retract.

2) Move the printwheel back to its position of visibility.

3) Insure that output signals have their "in between print cycles" status.

We now arrive at another fundamental programming concept: **there is e "most efficient" sequence in which you should write assembly language source programs.** We

PROGRAM IMPLEMENTATION SEQUENCE

could go ahead and write an initialization program to implement a RESET, but that would require a lot of guessing. How do we know that the printhammer has been fired? How do we move the printwheel back to its position of visibility? RESET is going to abort a print cycle — therefore the print cycle program must be created before we can know how to abort it.

Generally stated, you should start writing a program by implementing the most important event in your logic, then you should work eway from this beginning, implementing dependent events.

Specifically, we are going to postpone creating a program to implement the RESET logic until the print cycle program has been created.
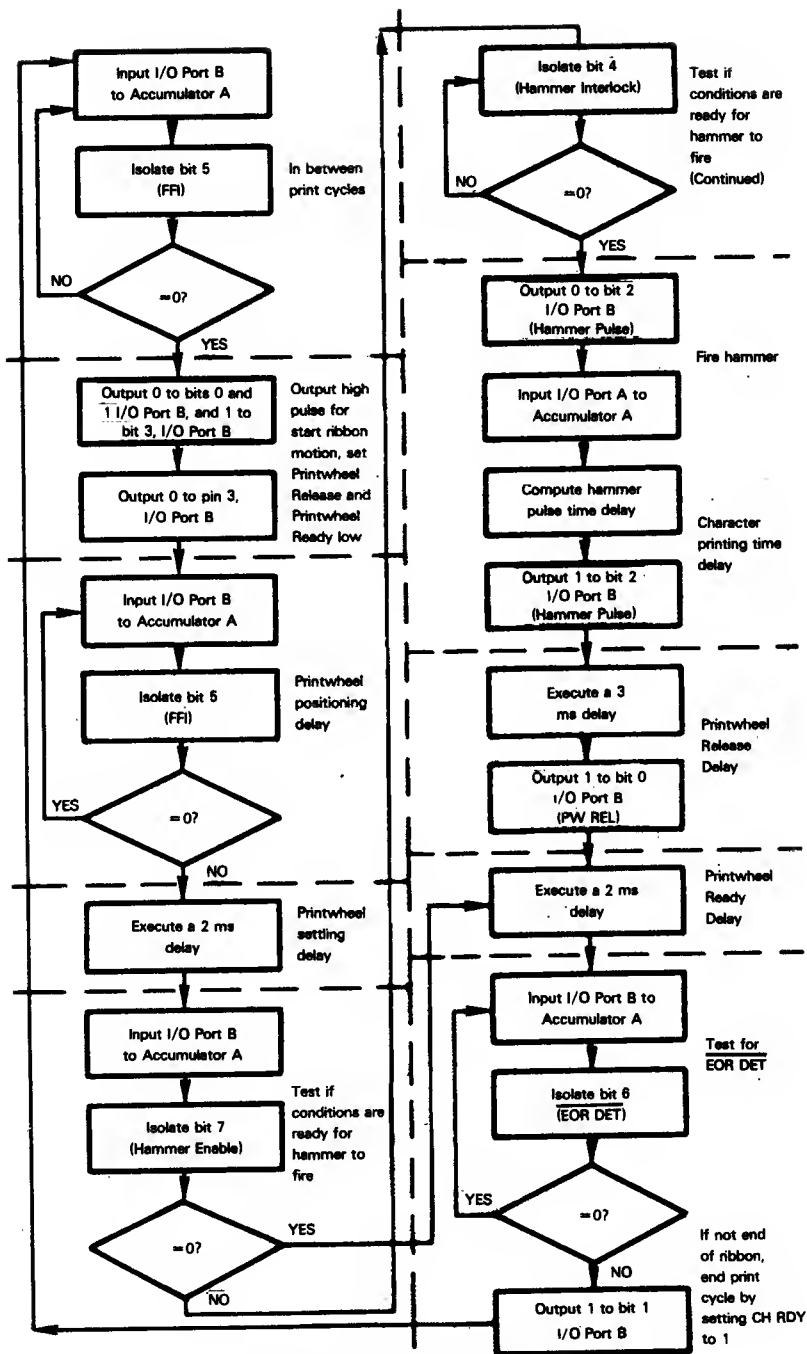
Figure 4-3. First Attempt At Program Flowchart

PROGRAM FLOWCHART

Let us now turn our attention to the functions which must be performed by the microcomputer system. These functions are identified by the flowchart illustrated in Figure 4-3. We will analyze this flowchart, step-by-step.

We are going to use the velocity decode input signal (FFI) to identify the start of a new print cycle. In between print cycles, therefore, the program continuously inputs I/O Port B contents to Accumulator A, testing bit 5. So long as this bit equals 1, a new print cycle has not begun. As soon as this bit equals 0, a new print cycle is identified:
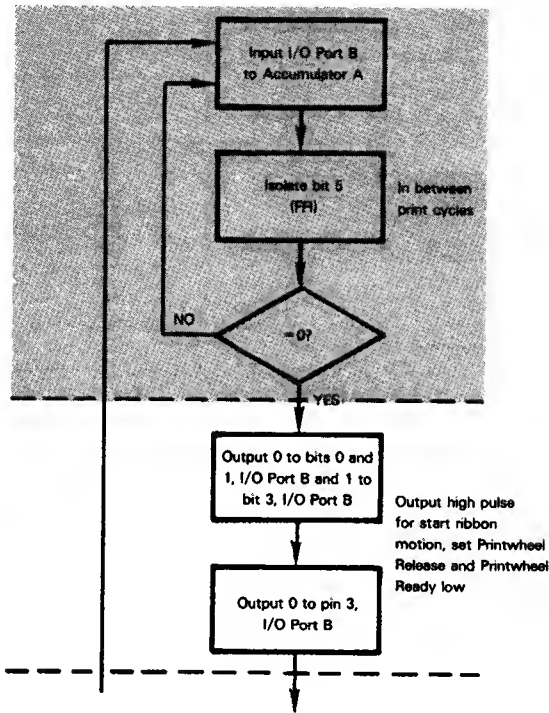
The first thing that happens within the new print cycle is that **a high START RIBBON MO-TION pulse is output by sequentially writing a 1, then a 0 to bit 3 of I/O Port B. Also, 0s are output to bits 0 and 1 of I/O Port B,** since PRINTWHEEL RELEASE and PRINTWHEEL READY must both be output low at the start of the print cycle:
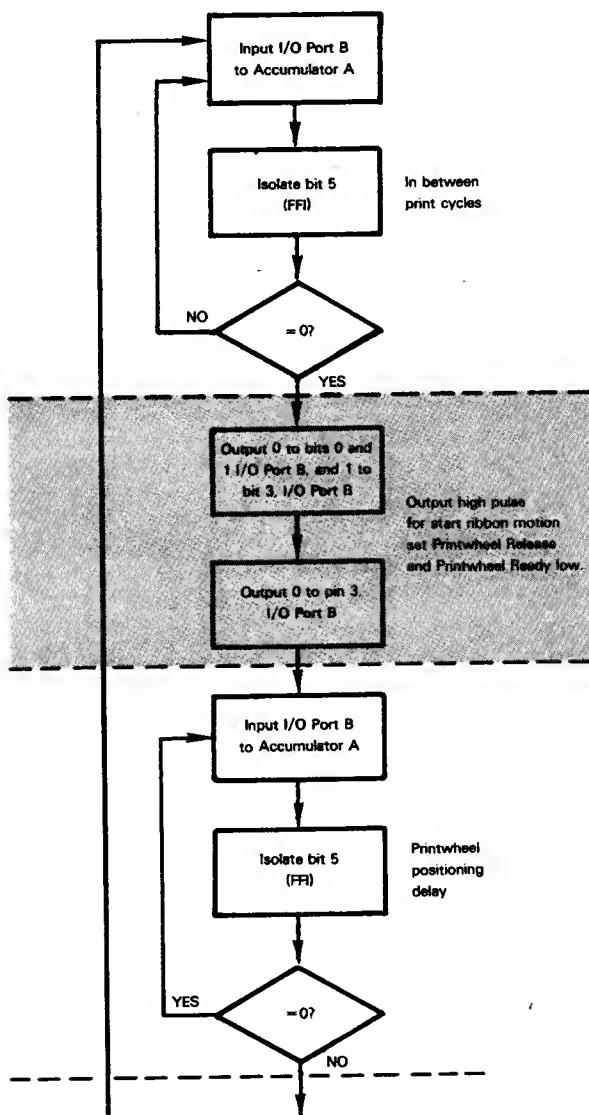
The printwheel positioning delay is computed by the velocity decode signal FFI. So long as this signal is low, the printwheel is still being positioned. We therefore go into a variable delay loop, which in terms of program logic is the inverse of the "in between print cycles" delay loop. Once again, I/O Port B contents are input to Accumulator A and bit 5 is tested; however, we stay in the delay loop until bit 5 is 1. At that time the printwheel positioning delay is over:

YES

Output 0 to bits 0 and 1 I/O Port B, and 1 to bit 3, I/O Port B

Output high pulse for start ribbon motion, set Printwheel Release and Printwheel Ready low

Output 0 to pin 6, I/O Port B

Input I/O Port B to Accumulator A

Isolate bit 5 (FFI)

Printwheel positioning delay

= 0?

YES

NO

Execute a 2 ms delay

Printwheel settling delay

4-12
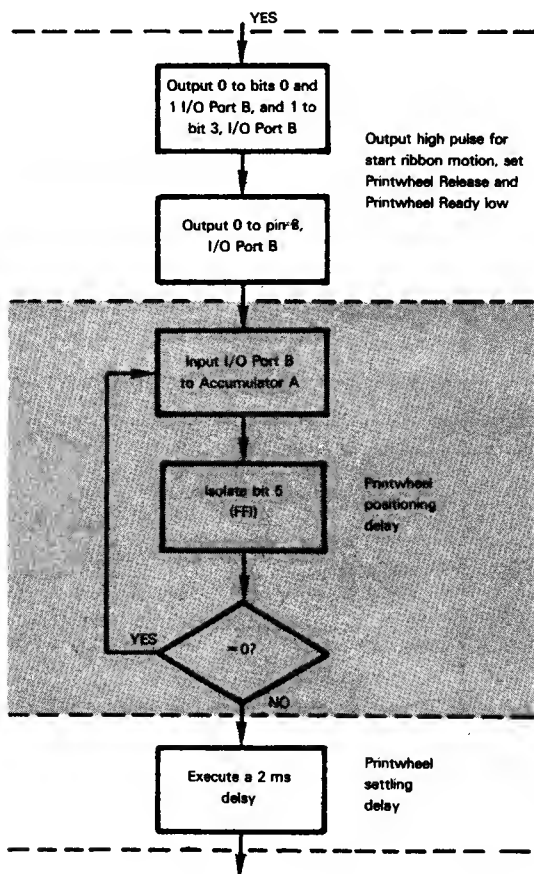
The printwheel positioning delay must be followed by a 2 millisecond printwheel settling delay. The usual delay loop will be executed here:

```
                    ┌──────────────────────┐
                    │   Input I/O Port B    │
                    │   to Accumulator A    │
                    └──────────────────────┘
                               │
                    ┌──────────────────────┐        Printwheel
                    │    Isolate bit 5      │        positioning
                    │        (FR)           │        delay
                    └──────────────────────┘
                               │
                          ╱─────────╲
                YES      ╱    = 0?    ╲
           ◄────────────╲             ╱
                          ╲─────────╱
                               │ NO
                    ┌──────────────────────┐        Printwheel
                    │    Execute a 2 ms     │        settling
                    │        delay          │        delay
                    └──────────────────────┘
                               │
                    ┌──────────────────────┐
                    │   Input I/O Port B    │
                    │   to Accumulator A    │
                    └──────────────────────┘
                               │
                    ┌──────────────────────┐        Test if conditions
                    │    Isolate bit 7      │        are ready for
                    │   (Hammer Enable)     │        hammer to fire
                    └──────────────────────┘
                               │
                          ╱─────────╲      YES
                         ╱    = 0?    ╲────────►
                         ╲             ╱
                          ╲─────────╱
                               │ NO
                               ▼
```
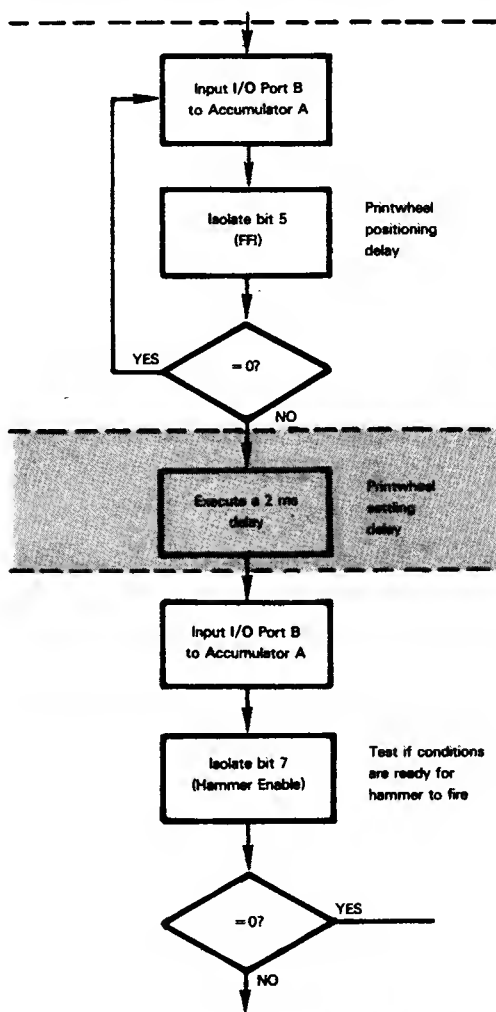
At the end of the printwheel settling delay, **the printhammer is fired, providing the HAMMER INTERLOCK signal is low and HAMMER ENABLE is high.** Recall that HAMMER INTERLOCK is a single status bit, used by all external conditions that can prevent the hammer from being fired. Any signal inputting a high level to this status pin will suppress printhammer firing.

4-13

A printwheel repositioning print cycle is identified by HAMMER ENABLE being input low. This condition is detected by isolating bit 7 of I/O Port B before testing the condition of HAMMER INTERLOCK. If bit 7 of I/O Port B equals 0, then the entire printhammer firing sequence is skipped and we jump directly to the printwheel ready delay, which is the last time delay of the print cycle:
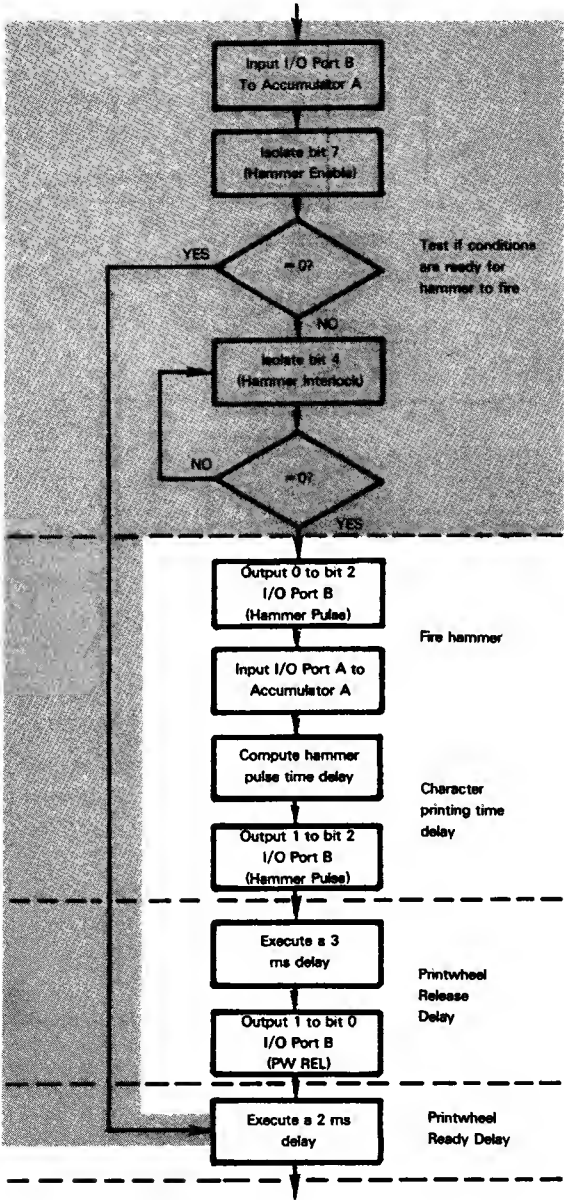
If HAMMER ENABLE is high, this is a character printing cycle, so the printhammer will be fired, but only when HAMMER INTERLOCK is 0. So long as any signal wire-ORed to pin 4 of I/O Port B is high, the program will stay in an endless loop, continuously testing the status of this I/O port pin. When finally the I/O port pin equals 0, the program will advance to the printhammer firing instruction sequence:
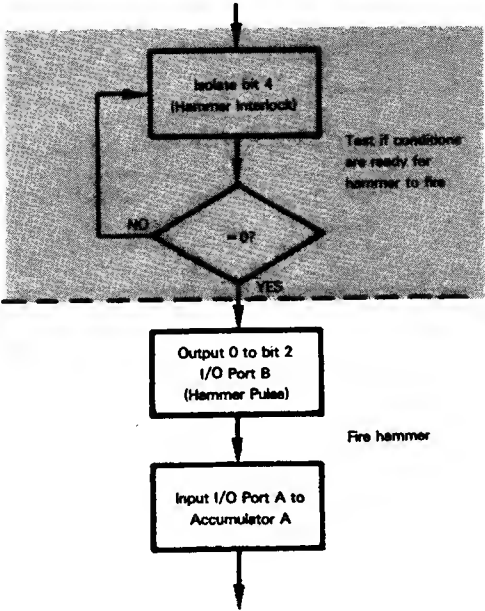
In order to fire the printhammer, a variable length firing pulse must be output. To do this a 0 is output to pin 2 of I/O Port B, since this is the pin via which the hammer pulse is output. Next the hammer pulse time delay is computed. We will describe how the hammer pulse width is computed after completing a description of the flowchart. At the end of the printhammer firing time delay, a 1 is output to bit 2 of I/O Port B. This terminates the printhammer firing pulse:

```
                  ┌────────────────┐
                  │  Isolate bit 4  │
              ┌──▶│ (Hammer Interlock)│
              │   └────────┬───────┘
              │            │              Test if conditions
              │            ▼              are ready for
          NO  │        ◇───────◇          hammer to fire
              └───────◇  = 0?  ◇
                       ◇───────◇
                           │ YES
                           ▼
                  ┌────────────────┐
                  │ Output 0 to bit 2│
                  │   I/O Port B    │
                  │  (Hammer Pulse) │
                  └────────┬───────┘
                           │              Fire hammer
                           ▼
                  ┌────────────────┐
                  │ Input I/O Port A to│
                  │  Accumulator A  │
                  └────────┬───────┘
                           │
                           ▼
                  ┌────────────────┐
                  │ Compute hammer  │
                  │ pulse time delay│
                  └────────┬───────┘
                           │              Character
                           ▼              printing time
                  ┌────────────────┐      delay
                  │ Output 1 to bit 2│
                  │   I/O Port B    │
                  │  (Hammer Pulse) │
                  └────────┬───────┘
                           │
                           ▼
                  ┌────────────────┐
                  │   Execute a 3   │
                  │    ms delay     │
                  └────────┬───────┘
                           │              Printwheel
                           ▼              Release
                  ┌────────────────┐      Delay
                  │ Output 1 to bit 0│
                  │   I/O Port B    │
                  │    (PW REL)     │
                  └────────┬───────┘
                           │
                           ▼
```
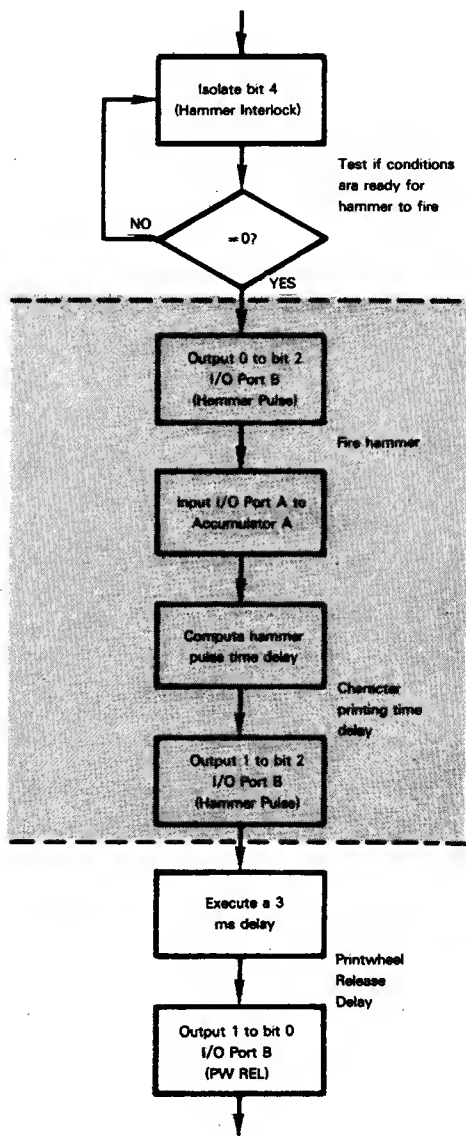
Now two settling delays follow. First there is a 3 millisecond printwheel release delay, the termination of which is marked by a 1 being output to bit 0 of I/O Port B. This causes PW REL to output high:

```
                    │
                    ▼
         ┌──────────────────┐
         │  Output 1 to bit 2 │   Character
         │     I/O Port B     │   printing time
         │   (Hammer Pulse)   │   delay
         └──────────────────┘
                    │
    ┌ ── ── ── ── ──┼── ── ── ── ── ┐
    │      ┌──────────────────┐      │
    │      │   Execute a 3    │      │
    │      │     ms delay     │      │
    │      └──────────────────┘      │
    │                │        Printwheel │
    │                │        Release    │
    │      ┌──────────────────┐ Delay    │
    │      │  Output 1 to bit 0 │        │
    │      │     I/O Port B     │        │
    │      │     (PW REL)       │        │
    │      └──────────────────┘      │
    └ ── ── ── ── ──┼── ── ── ── ── ┘
                    │
           ┌──────────────────┐
    ──────▶│   Execute a 2 ms  │   Printwheel
           │      delay        │   Ready Delay
           └──────────────────┘
    ── ── ── ── ── ──┼── ── ── ── ──
                    ▼
```
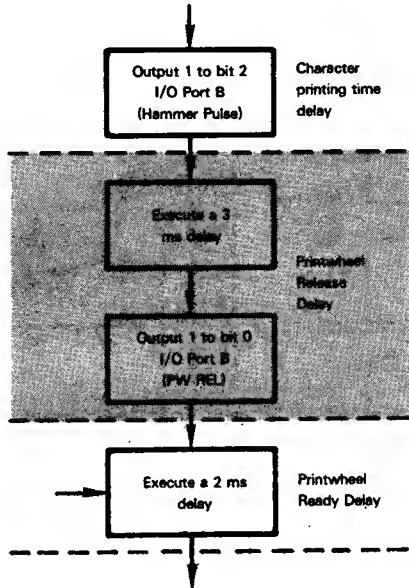
Next, a 2 millisecond printwheel ready delay is executed. The end of this delay and **the end of the print cycle is marked by a 1 output to bit 1 of I/O Port B;** this sets CH RDY high. **We do not want to do this, however, if there is an end-of-ribbon status.** This status is identified by $\overline{\text{EOR DET}}$ being low.

The program therefore inputs I/O Port B and isolates bit 6, via which $\overline{\text{EOR DET}}$ is input to the microcomputer system. If $\overline{\text{EOR DET}}$ equals 0, then the program stays in an endless loop, continuously retesting bit 6 of I/O Port B; thus another print cycle cannot begin. Only if $\overline{\text{EOR DET}}$ is detected equal to 1 will the print cycle terminate with CH RDY set to 1:
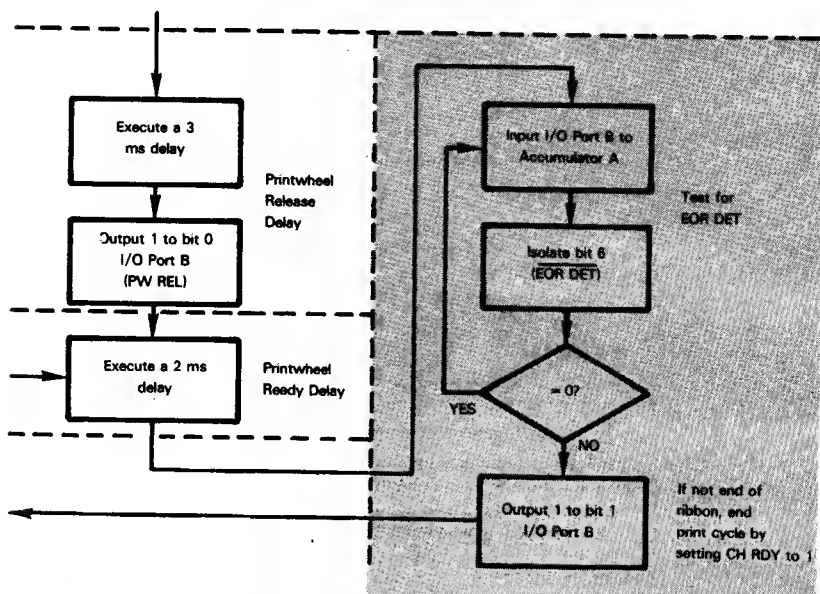
Now let us turn our attention to the method via which the appropriate printhammer firing delay is computed. In Figure 3-1, the appropriate printhammer firing delay was signaled by one of six lines (H1 through H6) being input true. Some external logic had to generate the true line, based on the nature of the character being printed; **this kind of operation is easier to do within a microcomputer program.**

PRINTHAMMER FIRING DELAY

This is the method we will use to compute the appropriate printhammer firing pulse time delay: every character to be printed is represented by one ASCII code data byte, as illustrated in Appendix A.

If we ignore the high order parity bit, then 128 possible bit combinations remain. If you look at the ASCII codes given in Appendix A, you will see that only character codes between 20_{16} and $7A_{16}$ are significant. Therefore, only $5A_{16}$ (or 90_{10}) code combinations need to be accounted for. Each of these code combinations will have assigned to it one byte in a 90-byte table; and in this byte will be stored a number between 1 and 6. This number will identify the time delay required by the character. A 12-byte table will contain the six actual time delays associated with the six digits. This scheme may be illustrated as follows:
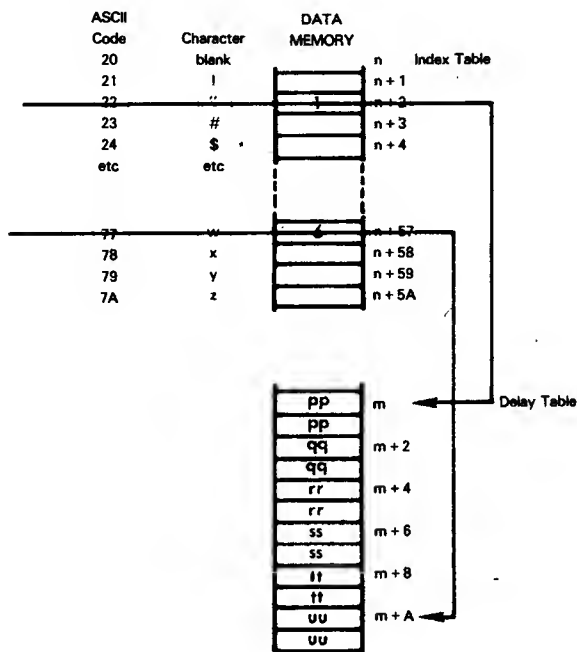
In the above illustration the letters "n" and "m", to the right of the data memory, represent any valid base memory addresses. For example, "n" might represent $FF80_{16}$ while "m" represents $FFF0_{16}$.

Consider two examples.

ASCII code 22_{16} signifies the double quotes character ("), which requires the shortest time delay. The data memory byte with address n + 2 corresponds to this ASCII code. 1 is stored in this data memory byte. Therefore, the first time delay, represented by pppp, is the value which must be loaded into the Index register before executing the long time delay loop which creates the printhammer firing pulse for the " character.

ASCII code 77_{16} represents "w". The data memory byte with address n + 57_{16} corresponds to this ASCII code. Within this data memory byte the value 6 is stored, which means that the longest printhammer firing delay is required for a "w". Therefore, a value represented by uuuu will be loaded into the Index register before executing the long time delay loop which creates the printhammer firing pulse for the w character.

Figure 4-4 identifies the program steps via which the printhammer firing delay will be computed.

In order to better understand Figure 4-4, we will go down steps Ⓐ through Ⓘ for the case of "w".

Ⓐ The ASCII representation of lower case w is input to Accumulator A:

From I/O Port A

A [X1110111]

Input ASCII character code from Port A	Ⓐ
Set bit 7 to 0	Ⓑ
Subtract 20₁₆	Ⓒ
Move to Index register	Ⓓ
Input from Index Table	Ⓔ

Multiply by 2	Ⓕ
Move to Index register	Ⓖ
Input delay to Index register	Ⓗ
Execute printhammer firing delay	Ⓘ
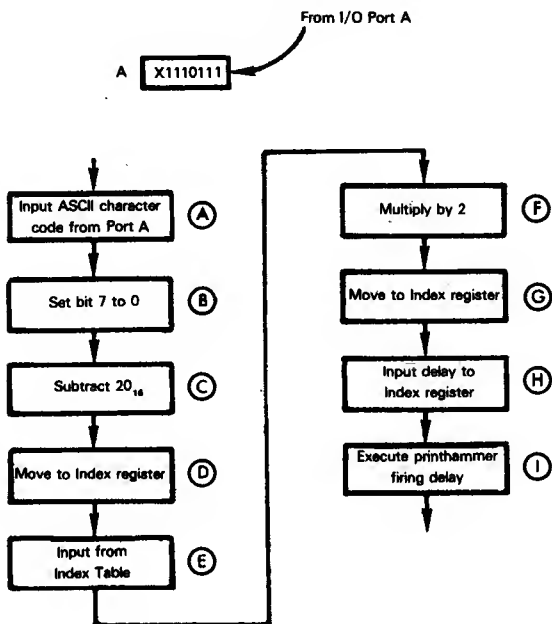
Figure 4-4. Program Flowchart To Compute Printhammer
Firing Pulse Length

Ⓑ We must set the parity bit to 0. To do this Accumulator A contents are ANDed with $7F.

A [X1110111]

X1110111
01111111
————————
01110111
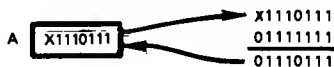
Ⓒ The Index Table entry corresponding to lower case w is computed by adding the ASCII code, less 20₁₆ to the Index Table base address. We must subtract 20₁₆ because the first 1F codes have no ASCII equivalent.

A [01110111]

01110111
11100000
————————
01010111

Twos complement of 20₁₆
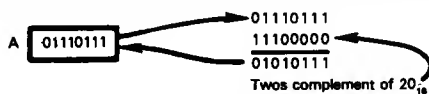
Ⓓ The Accumulator A contents are moved to the Index register:

A [01010111]

IX [1111111101010111]
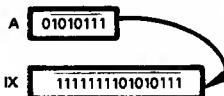
(E) Using direct, indexed addressing we can now load the required time delay index from the Index Table to Accumulator A. Assuming the Index Table base address is FF80$_{16}$, this may be illustrated as follows:
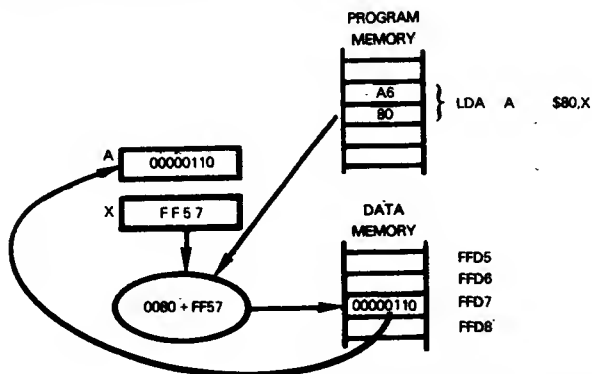
(F) Since the actual delay is two bytes long, we are going to calculate the address of the appropriate delay by adding twice the index to the Delay Table base address. First we multiply the index, which is currently in Accumulator A, by 2:

A | 00001100

(G) The Accumulator A contents are again moved to the Index register:

A | 00001100
IX | 1111111100001100
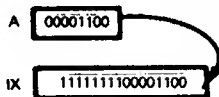
(H) Using direct, indexed addressing, the 16-bit delay constant is loaded into the Index register. Assuming that the Delay Table base address is FFF0$_{16}$, this may be illustrated as follows:
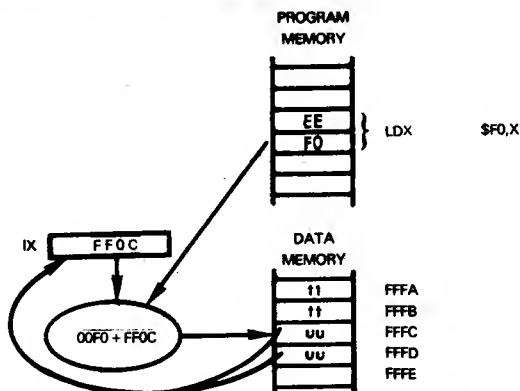
(I) The Index register now contains the correct initial value for a long delay to be executed as described in Chapter 2.

Putting together the program flowcharts illustrated in Figures 4-3 and 4-4, we generate the entire required program, as illustrated in Figure 4-5. This program is now described, section-by-section.

```
PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O PORT B) FOR A 0 VALUE
START   LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$20        ISOLATE BIT 5
        BNE     START       IF NOT 0, RETURN TO START
INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF I/O PORT B. OUTPUT 1 TO BITS 2
AND 3 OF I/O PORT B.
        LDA A   #$C         LOAD MASK INTO ACCUMULATOR A
        STA A   $C001       OUTPUT TO I/O PORT B
OUTPUT 0 TO BIT 3 OF I/O PORT B. THIS COMPLETES START RIBBON MOTION PULSE
        LDA A   #4          LOAD MASK INTO ACCUMULATOR A
        STA A   $C001       OUTPUT TO I/O PORT B
TEST FOR END OF PRINTWHEEL POSITIONING. BIT 5 OF I/O PORT B (FFI) WILL BE 1
LOP1    LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$20        ISOLATE BIT 5
        BEQ     LOP1        IF 0 RETURN TO LOP1
EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        LDX     #$FA        LOAD INITIAL TIME DELAY CONSTANT
LOP2    DEX                 DECREMENT INDEX REGISTER
        BNE     LOP2        RE-DECREMENT IF NOT ZERO
TEST PRINTHAMMER FIRING CONDITIONS
LOP3    LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ROL A               MOVE BIT 7 INTO CARRY
        BCC     PRD         IF CARRY IS ZERO BYPASS PRINTHAMMER FIRING
        AND A   #$20        ISOLATE BIT 4 WHICH IS NOW BIT 5
        BEQ     LOP3        WAIT FOR NONZERO VALUE BEFORE FIRING
FIRE PRINTHAMMER
        LDA A   $C001       SET HAMMER PULSE LOW. OUTPUT 0
        AND A   #$FB        TO BIT 2 OF I/O PORT B
        STA A   $C001
        LDA A   $C000       INPUT ASCII CHARACTER TO ACCUMULATOR A
        AND A   #$7F        MASK OUT HIGH ORDER BIT
        SUB A   #$20        SUBTRACT $20
        STA A   SCRA + 1    MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX     SCRA
        LDA A   INDEX,X     LOAD INDEX INTO ACCUMULATOR A
        ASL A               MULTIPLY BY 2
        STA A   SCRA + 1    MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX     SCRA
        LDX     DELY,X      LOAD DELAY CONSTANT INTO INDEX REGISTER
LOP4    DEX                 EXECUTE LONG DELAY
        BNE     LOP4
        LDA A   $C001       AT END OF DELAY OUTPUT 1 TO BIT 2
        ORA A   #4          OF I/O PORT B. THIS SETS HAMMER PULSE HIGH
        STA A   $C001
EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
        LDX     #374        LOAD INITIAL TIME DELAY CONSTANT
LOP5    DEX                 EXECUTE LONG TIME DELAY
        BNE     LOP5
OUTPUT 1 TO BIT 0 OF I/O PORT B. THIS SETS PW REL HIGH
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #1          SET BIT 0 TO 1
        STA A   $C001       OUTPUT RESULT
```

EXECUTE A 2 MS PRINTWHEEL READY DELAY

```
PRD     LDX     #$FA        LOAD INITIAL TIME DELAY
LOP6    DEX                 DECREMENT INDEX REGISTER
        BNE     LOP6        RE-DECREMENT IF NOT ZERO
```

TEST FOR EOR DET (BIT 6 OF I/O PORT B) EQUAL TO 0 AS A PREREQUISITE FOR ENDING THE
PRINT CYCLE

```
LOP7    LDA  A  $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND  A  #$40        ISOLATE BIT 6
        BEO     LOP7        RETURN AND RETEST IF 0
```

AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT B TO 1
THIS SETS CH RDY HIGH

```
        LDA  A  $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA  A  #2          SET BIT 1 TO 1
        STA  A  $C001       OUTPUT RESULT
        JMP     START       JUMP TO NEW PRINT CYCLE TEST
```

Figure 4-5. A Simple Print Cycle Instruction Sequence
Without Initialization Or Reset

**In between print cycles the following three-instruction loop continuously tests
the status of I/O Port B, bit 5.** The FFI signal is input to this pin. So long as this signal is input
high, a new print cycle cannot start. As soon as this signal is input low, the printwheel is identified
as being in motion — which means that a new print cycle is underway:

PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O PORT B) FOR A 0 VALUE

```
Enter     START   LDA  A   $C001    INPUT I/O PORT B TO ACCUMULATOR A
Program           AND  A   #$20     ISOLATE BIT 5
                  BNE      START    IF NOT 0, RETURN TO START
```

INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF I/O PORT B. OUTPUT 1 TO BITS 2
AND 3 OF I/O PORT B

```
                  LDA  A   #$C      LOAD MASK INTO ACCUMULATOR A
```

**As soon as a new print cycle starts, the PRINTWHEEL RELEASE and PRINT-
WHEEL READY signals must be output low. Also, a high START RIBBON MOTION
pulse must be output** so that when the printhammer fires, fresh ribbon is in front of the
character which is to be printed. These initial signal changes may be illustrated as follows:

INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF I/O PORT B. OUTPUT 1 TO BITS 2
AND 3 OF I/O PORT B.

```
        LDA  A  #$C         LOAD MASK INTO ACCUMULATOR A
        STA  A  $C001       OUTPUT TO I/O PORT B
```

OUTPUT 0 TO BIT 3 OF I/O PORT B. THIS COMPLETES START RIBBON MOTION PULSE

```
        LDA  A  #4          LOAD MASK INTO ACCUMULATOR A
        STA  A  $C001       OUTPUT TO I/O PORT B
```


MC6820
Port B

```
3   START RIBBON MOTION
2   HAMMER PULSE
1   PRINTWHEEL READY
0   PRINTWHEEL RELEASE
```
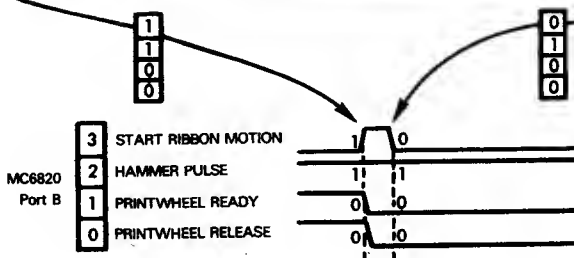
In the above illustration, notice that I/O Port B, pin 2 has been forced to output 1. This is the $\overline{\text{HAMMER PULSE}}$ pin, which goes low only for the duration of the printhammer firing pulse. At this point in the print cycle, this signal is high, so outputting 1 is harmless.

The program now executes a variable length delay, during which time the printwheel either moves until the appropriate character petal is in front of the printhammer, or the print-wheel moves back to its position of visibility. In either case exter-

nal logic inputs signal FFI low for the duration of the printwheel positioning delay. As soon as the printwheel has been positioned, FFI is detected high — and program **logic advances to the 2 millisecond printwheel settling delay.** We have seen this three-instruction delay loop frequently before:

```
           TEST FOR END OF PRINTWHEEL POSITIONING. BIT 5 OF I/O PORT B (FFI) WILL BE 1
    LOP1    LDA A    $C001        INPUT I/O PORT B TO ACCUMULATOR A
            AND A    #$20         ISOLATE BIT 5
            BEQ      LOP1         IF 0 RETURN TO LOP1
           EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
            LDX      #$FA         LOAD INITIAL TIME DELAY CONSTANT
    LOP2    DEX                   DECREMENT INDEX REGISTER
            BNE      LOP2         RE-DECREMENT IF NOT ZERO
           TEST PRINTHAMMER FIRING CONDITIONS
```
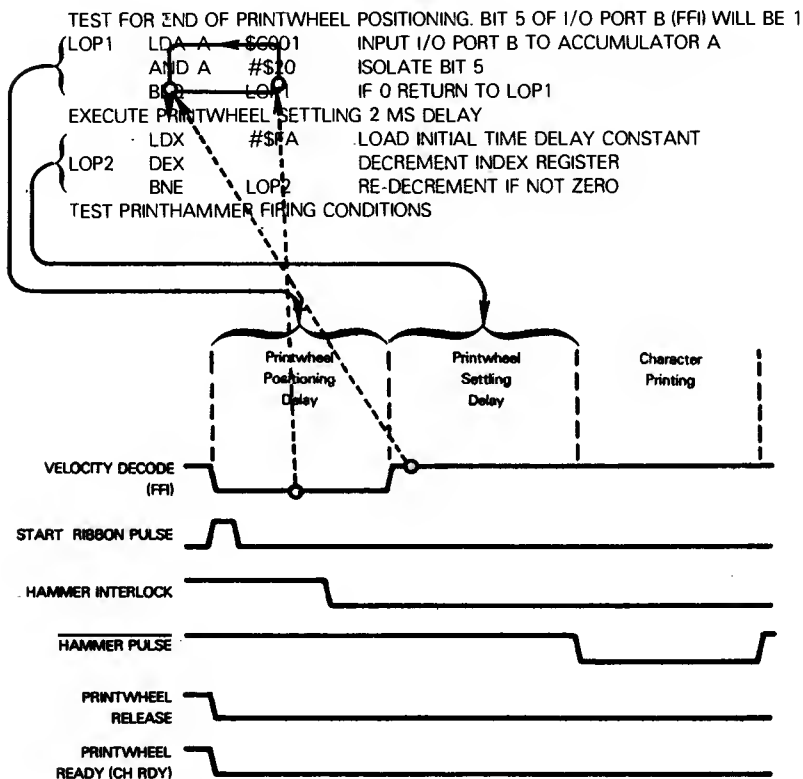

Now the printhammer is ready to be fired. First we test the condition of $\overline{\text{HAMMER ENABLE}}$, which has been connected to pin 7 of I/O Port B. If this signal is low, then we are in a printwheel repositioning print cycle and the entire hammer firing instruction sequence is bypassed. Notice that the condition of bit 7 is tested by shifting into the Carry status. **If $\overline{\text{HAMMER ENABLE}}$ is high, we pass this test. But HAMMER INTERLOCK must still be tested;** this signal is input to I/O Port B, pin 4.

The Shift instruction moved bit 4, (representing the HAMMER INTERLOCK) to bit 5, and bit 7 (representing HAMMER ENABLE) into the Carry status:
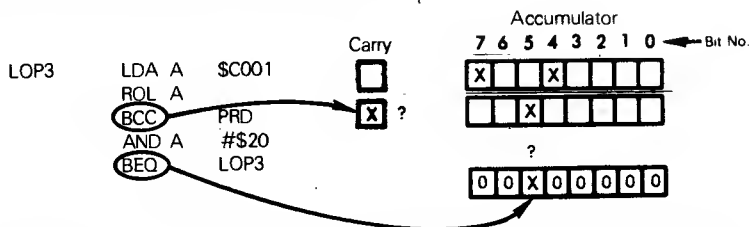
Having loaded the contents of I/O Port B into Accumulator A once, we have serially tested the condition of two bits. Each bit could have been tested individually via the following six instructions:

```
       LDA A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
       AND A    #$80     ISOLATE BIT 7
       BEQ      PRD      IF BIT 7 IS 0 BYPASS PRINTHAMMER FIRING
LOP3   LDA A    $C001    INPUT I/O PORT B TO ACCUMULATOR A
       AND A    #$10     ISOLATE BIT 4
       BEQ      LOP3     WAIT FOR NONZERO VALUE BEFORE FIRING
```

If HAMMER ENABLE is detected low, execution branches to the instruction labeled PRD. You will find this instruction close to the end of the program, at the beginning of the instruction sequence which executes **a 2 millisecond PRINTWHEEL READY delay.**

Note that the five-instruction sequence illustrated in Figure 4-5 tests for HAMMER ENABLE low within the loop that tests for HAMMER INTERLOCK high. Now HAMMER ENABLE will be either high or low for the duration of the print cycle; it will not change level during the print cycle. Therefore the fact that it is continuously being tested is redundant — it serves no purpose, but it does no harm.

Next the printhammer is fired. The instruction sequence which causes the printhammer to fire implements steps Ⓐ through Ⓘ , which we have already described. In order to make the instruction sequence easier to understand, it is reproduced below with labels Ⓐ through Ⓘ added:

```
FIRE PRINTHAMMER
             LDA A    $C001    SET HAMMER PULSE LOW. OUTPUT 0
             AND A    #$FB     TO BIT 2 OF I/O PORT B
             STA A    $C001
Ⓐ           LDA A    $C000    INPUT ASCII CHARACTER TO ACCUMULATOR A
Ⓑ           AND A    #$7F     MASK OUT HIGH ORDER BIT
Ⓒ           SUB A    #$20     SUBTRACT $20
Ⓓ         { STA A    SCRA + 1 MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
            { LDX      SCRA
Ⓔ           LDA A    INDEX,X  LOAD INDEX INTO ACCUMULATOR A
Ⓕ           ASL A             MULTIPLY BY 2
Ⓖ         { STA A    SCRA + 1 MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
            { LDX      SCRA
Ⓗ           LDX      DELY,X   LOAD DELAY CONSTANT INTO INDEX REGISTER
LOP4      { DEX               EXECUTE LONG DELAY
Ⓘ         { BNE      LOP4
             LDA A    $C001    AT END OF DELAY OUTPUT 1 TO BIT 2
             ORA A    #4       OF I/O PORT B. THIS SETS HAMMER PULSE HIGH
             STA A    $C001
EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
```

Notice that in order to transfer the contents of Accumulator A to the Index register, we reserve two bytes of memory as a scratch read/write area:

STA A   SCRA + 1
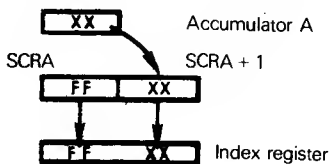
LDX   SCRA

We assume that SCRA contains the value FF_{16}.

Why are we loading data into Accumulator A, then transferring it to the Index register? The reason is that data contributes to the address computation only if it is stored in the Index register. While data resides in Accumulator A or Accumulator B, it cannot be used as part of the address computation. So why did we not load the data into the Index register in the first place? There are two reasons:

1) You cannot perform arithmetic operations upon the contents of the Index register; and we need to perform such operations before using the data as part of the address computation.

2) The Index register loads two bytes of data at a time. But we are loading data from an MC6820 I/O port. Were we to load data into the Index register, we would get the contents of I/O Port B and Control Register A as follows:
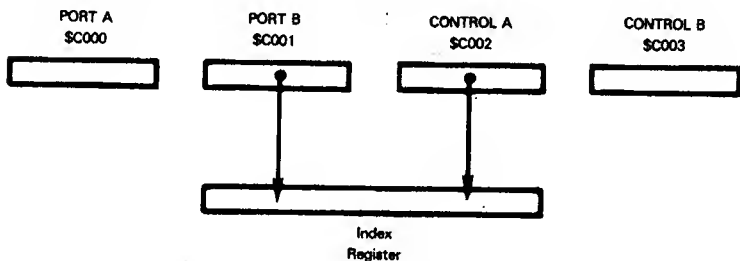
Therefore, we have to load the contents of the I/O port into an Accumulator, then move it to the Index register.

A 3 millisecond PRINTWHEEL RELEASE time delay is now executed and the end of this time delay is marked by the PRINTWHEEL RELEASE signal being output high. Next, the 2 millisecond PRINTWHEEL READY delay is executed:

```
        EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
              LDX      #374      LOAD INITIAL TIME DELAY CONSTANT
   LOP5       DEX                EXECUTE LONG TIME DELAY
              BNE      LOP5
        OUTPUT 1 TO BIT 0 OF I/O PORT B. THIS SETS PW REL HIGH
              LDA A    $C001     INPUT I/O PORT B TO ACCUMULATOR A
              ORA A    #1        SET BIT 0 TO 1
              STA A    $C001     OUTPUT RESULT
        EXECUTE A 2 MS PRINTWHEEL READY DELAY
   PRD        LDX      #$FA      LOAD INITIAL TIME DELAY
   LOP6       DEX                DECREMENT INDEX REGISTER
              BNE      LOP6      RE-DECREMENT IF NOT ZERO
```
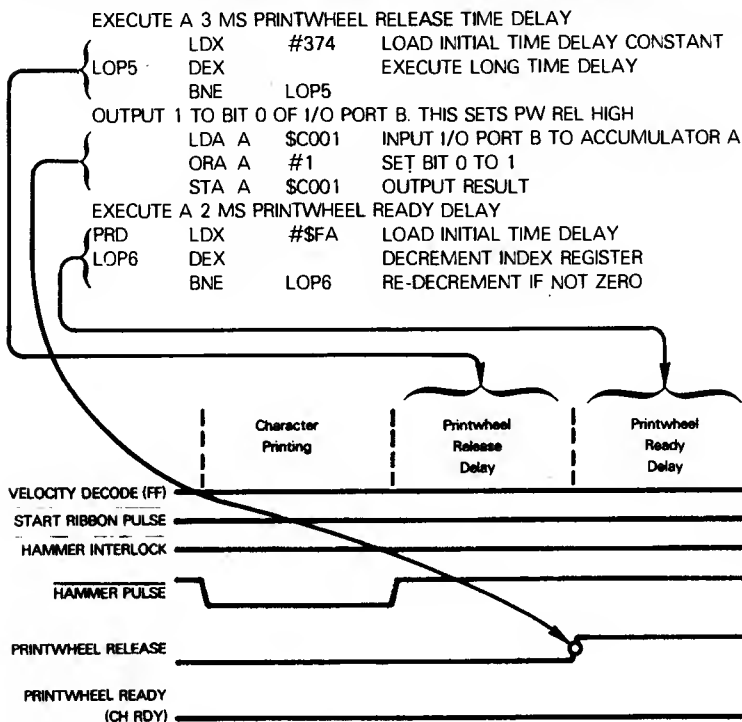

Before terminating the print cycle by outputting PRINTWHEEL READY (CH RDY) high, the program must insure that the end of ribbon has not been reached. If EOR DET is detected low the program stays in an endless loop until the ribbon has been changed; then EOR DET will be input high by external logic.

When EOR DET is detected high, the final instructions of the program set PRINTWHEEL READY high, then return to the beginning of the program and wait for the next print cycle.

PROGRAM LOGIC ERRORS

The program we have developed in this chapter contains a logic error which could not occur in a digital logic implementation. The error is in the hammer pulse time delay computation.

In a digital logic implementation, the ASCII code for any character would be processed as seven individual signals. These signals would be combined in some way to generate one of the time delay signals H1 through H6. **It does not matter what ASCII code combination is input, one of the time delay signals H1 through H6 will be output high;** if the signal generation logic is unsound, a time delay signal will still be created, although it may be the wrong signal.

Now look at the assembly language program implementation. It is simple enough for us to look up the table in Appendix A and see that valid ASCII codes only cover the range 20_{16} through $7A_{16}$. That does not prevent a logic designer from using the microcomputer system we create in a special system that includes unusual characters, represented by codes outside the normal ASCII range. Our program could output some very strange results under these circumstances. Suppose the ASCII code 10_{16} had been adopted to represent a special character. Then, our attempt to look up the Index Table would load into Accumulator A whatever happened to be in memory byte $n-10_{16}$.

`LIMIT CHECKING`

There is no telling what could be in this memory byte; in all probability, this byte will be used to store an instruction code, perhaps a two-hexadecimal-digit value. Suppose it contained $2A_{16}$; the next program step will double $2A_{16}$, add it to the base address of the Delay Table and access the initial delay code from memory location $m + 54_{16}$.

Given the microcomputer configuration illustrated in Figure 4-2, this memory location could easily be one of the duplicate addresses which spuriously access some memory byte, because we have used disarmingly simple chip select logic. Had we used more complex chip select logic, then chances are we would now be attempting to access a memory byte that did not exist. In the former case, there is no telling what length of hammer pulse would be generated; in the latter case, an extremely long hammer pulse would be generated, since we would retrieve 0 from a non-existent memory location, and this value would be interpreted as the initial delay constant for the long delay program loop. The hammer pulse would be 524 milliseconds long:

$$65,536 \times 8 = 524,288 \text{ microseconds}$$

Time in microseconds to execute long delay loop once.

Since initial 0000_{16} value will be decremented and then tested, a maximum delay loop results.
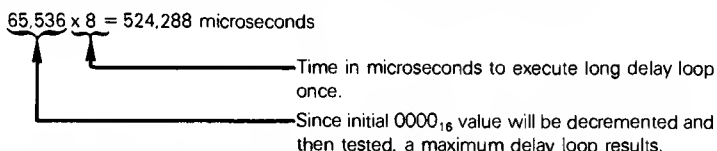
Now in order to avoid this problem we have two options:

1) Program logic can simply ignore any invalid ASCII code.

2) Program logic can generate a default hammer pulse width for invalid ASCII codes.

If we ignore special characters, the conclusion is obvious: the microcomputer system cannot be used in any application that requires special characters to be printed. Since the special character is ignored, nothing will happen when such a character code is detected on input --- there will be no hammer pulse, no carriage movement and no positioning.

Providing a default hammer pulse for special characters means that such characters will be printed, but they may create unevenness in the density of the typed text.

You, as the logic designer, would have to specify your preference.

Either instruction sequence may be inserted into the existing program as follows:

	STA A	$C001	
	LDA A	$C000	INPUT ASCII CHARACTER TO ACCUMULATOR A
Check for →	AND A	#$7F	MASK OUT HIGH ORDER BIT
valid ASCII	SUB A	#$20	SUBTRACT $20
codes inserted	STA A	SCRA + 1	MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
here			
	LDX	SCRA	
	LDA A	INDEX,X	LOAD INDEX INTO ACCUMULATOR A
	ASL A		MULTIPLY BY 2

Here is the instruction sequence which ignores non-standard ASCII codes:

```
           -
           -
           -
        LDA A    $C000    INPUT ASCII CHARACTER TO ACCUMULATOR A
        AND A    #$7F     MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CMP A    #$20
        BLT      PRD      IF CODE IS $1F OR LESS, BYPASS HAMMER FIRING
COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CMP A    #$7A
        BGT      PRD      IF CODE IS $7B OR GREATER, BYPASS HAMMER FIRING
ASCII CODE IS VALID
        SUB A    #$20     SUBTRACT $20
  .      -
         -
         -
```

The second option, illustrated below, prints unknown characters with a median density, using density code 3:

```
           -
           -
           -
        LDA A    $C000    INPUT ASCII CHARACTER TO ACCUMULATOR A
        AND A    #$7F     MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH SMALLEST LEGAL VALUE
        CMP A    #$20
        BGE      OK       IF CODE IS $20 OR MORE, TEST FOR HIGH LIMIT
CODE IS ILLEGAL, ASSUME A DENSITY OF 3
NOK     LDA A    #6       LOAD TWICE THE DENSITY
        JMP      NEXT
COMPARE ASCII CODE WITH LARGEST LEGAL VALUE
OK      CMP A    #$7A
        BGT      NOK      IF CODE IS $7B OR GREATER, ASSUME A DENSITY OF 3
ASCII CODE IS VALID
        SUB A    #$20     SUBTRACT $20
        STA A    SCRA + 1 MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX      SCRA
        LDA A    INDEX,X  LOAD INDEX INTO ACCUMULATOR A
        ASL A             MULTIPLY BY 2
NEXT    STA A    SCRA + 1 MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
           -
           -
           -
```

Both of the invalid ASCII code instruction sequences are simplistic in their solution to the problem.

The only new feature introduced is the use of the Compare Immediate (CMP A #) instruction. This instruction subtracts the immediate data in the operand from the contents of Accumulator A. The result of the subtraction is discarded, which means that the Accumulator contents are not altered; however, status flags are set to reflect the results of the subtraction. We use a BLT (Branch if Less Than) instruction to identify a

| COMPARE |
| IMMEDIATE |
| BRANCH ON |
| CONDITION |

negative result, which means that the immediate data in the operand was larger than the value in Accumulator A. Similarly, a BGT (Branch if Greater Than) instruction identifies a value in the immediate operand which is less than the contents of Accumulator A.

In the second instruction sequence, if the value in the immediate operand is less than, or equal to the contents of Accumulator A, the BGE instruction causes a branch to a later instruction labeled OK. The actual program execution paths for the second instruction sequence may appear a trifle confusing to you if you are new to programming; we therefore **illustrate execution paths as follows:**

```
        LDA  A    $C000      INPUT ASCII CHARACTER TO ACCUMULATOR A
        AND  A    #$7F       MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH SMALLEST LEGAL VALUE
  (A)   CMP  A    #$20
        BGE (B)   OK         IF CODE IS $20 OR MORE, TEST FOR HIGH LIMIT
CODE IS ILLEGAL. ASSUME A DENSITY OF 3
NOK     LDA  A    #$6        LOAD TWICE THE DENSITY
        JMP       NEXT  (E)
COMPARE ASCII CODE WITH LARGEST LEGAL VALUE
OK      CMP  A    #$7A
  (D)   BGT       NOK        IF CODE IS $7B OR GREATER, ASSUME A DENSITY OF 3
ASCII CODE IS VALID
        SUB  A    #$20       SUBTRACT $20
 (F) STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX       SCRA
  (C)   LDA  A    INDEX,X    LOAD INDEX INTO ACCUMULATOR A
        ASL  A               MULTIPLY BY 2
NEXT    STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
```

Execution paths, illustrated by circled letters above, can be interpreted as follows:

(A) An ASCII code passes the "lowest legal value" test, but now must be tested for the "highest legal value".

(B) The ASCII code failed the "lowest legal value" test. The program loads twice the default density into the Accumulator and branches to the instruction sequence which accesses the delay constant appropriate to this default density. This Jump is illustrated by (C)

(D) A character which has passed the "lowest legal ASCII value" test is next checked for "highest legal ASCII value"; if it fails this test then program execution branches, as shown by (E) , to instructions which assume a default density of 3. (E) , in fact, meets (B) .

(F) An ASCII character that passes both the "lowest legal value" test and the "highest legal value" test is processed via instruction path (F) . Instructions in this path load the appropriate density index into Accumulator A.

RESET AND INITIALIZATION

In order to complete our program, we must create the necessary Reset and Initialization instructions.

Reset instructions will be executed whenever RESET is input true to the microcomputer system.

Initialization instructions will be executed whenever the system is started up.

There is no reason why Reset and Initialization instruction sequences should coincide; in many applications two separate and distinct instruction sequences may be needed. **On the other hand, it is quite common to use Reset in lieu of system initialization.** This means that when you first power up the system, RESET is pulsed true; and this starts the entire microcomputer-based logic system.

In our case the Reset program is indeed simple. All we have to do is output Control codes to the MC6820 Peripheral Interface Adapter, then set output signals to the "in between

print cycles" condition. In addition, we must set read/write memory location SCRA for use in the printhammer firing sequence. **Here is the necessary Initialization instruction sequence:**

```
          ORG       $FC00
SYSTEM RESET AND INITIALIZATION
FIRST OUTPUT CONTROL CODE TO I/O PORT A CONTROL REGISTER
          CLR       $C002
NEXT OUTPUT CONTROL CODE TO I/O PORT B CONTROL REGISTER
          CLR       $C003
SET I/O PORT A TO INPUTS ONLY BY OUTPUTTING 0 TO DATA
DIRECTION REGISTER
          CLR       $C000
SET I/O PORT B TO OUTPUT VIA PINS 0 THROUGH 3 AND TO INPUT
VIA PINS 4 THROUGH 7
          LDA  A    #$0F
          STA  A    $C001
NOW OUTPUT CONTROL CODES THAT SELECT I/O PORTS A AND B
          LDA  A    #4
          STA  A    $C002
          STA  A    $C003
SET HAMMER PULSE, PW READY AND PW REL HIGH
SET START RIBBON MOTION LOW
          LDA  A    #7
          STA  A    $C001
LOAD $FF INTO UPPER BYTE OF SCRATCH READ/WRITE AREA
          CLR       SCRA
          COM       SCRA
```

This is how Control codes for each I/O port of the MC6820 PIA are initially constructed:
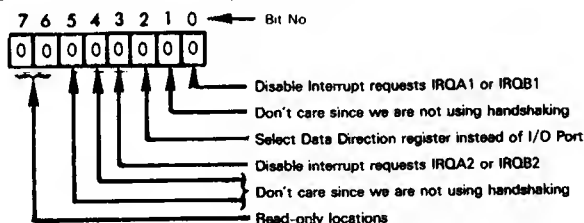
If the microcomputer system is Reset using the RESET input signal, then you will not have to output 0 Control codes, nor will you have to output 0 to a Data Direction register; the Reset operation will automatically zero all internal registers of an MC6820 PIA. If you are allowing for any type of programmed restart that does not use the Reset control signal, then you will have to output 0 to appropriate Control registers and Data Direction registers since you have no way of knowing what might have been in these registers previously.

Recall that the addresses $C000_{16}$ and $C001_{16}$ serve a double purpose. These addresses may access an I/O port, or they may access the Data Direction register associated with the I/O port. It is the condition of the associated Control register, bit 2, which determines whether the I/O port or Data Direction register will be selected. Thus after loading appropriate codes into the two Data Direction registers, we must output Control codes with 1 in bit 2, to Control registers for both I/O Port A and I/O Port B.

The codes output to the Data Direction registers cause all pins of I/O Port A to act as inputs. Recall that 0 in any bit position of a Data Direction register causes the associated I/O port pin to act as an input pin. A 1 in any bit position causes the associated I/O port pin to act as an output pin. Thus the four high order pins of I/O Port B are assigned to handle data input; the four low order pins are assigned to handle data output.

A PROGRAM SUMMARY

First of all, it would be a good idea to put together the entire program, as developed in this chapter. We will include the necessary Assembler directives. This final program is illustrated in Figure 4-6.

Here is the final program memory map identifying the way in which the program illustrated in Figure 4-6 uses ROM memory:

```
                    PROGRAM
                    MEMORY
                              FC00

                    Object
                    Program
                    Storage
                              FF7F
                    Index     FF80
                    Table     FFEF
                              FFF0
                    Delays    FFFF
                    Table
```
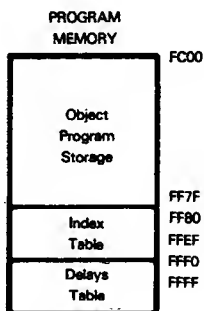
```
INDEX   EQU     $80         EQUATE INDEX TABLE
DELY    EQU     $EE         EQUATE DELAY TABLE BASE ADDRESS-2
SCRA    EQU     $0000       EQUATE SCRATCH AREA
        ORG     $FC00
SYSTEM RESET AND INITIALIZATION
FIRST OUTPUT CONTROL CODE TO I/O PORT A CONTROL REGISTER
        CLR     $C002
NEXT OUTPUT CONTROL CODE TO I/O PORT B CONTROL REGISTER
        CLR     $C003
SET I/O PORT A TO INPUTS ONLY BY OUTPUTTING 0 TO DATA
DIRECTION REGISTER
        CLR     $C000
SET I/O PORT B TO OUTPUT VIA PINS 0 THROUGH 3 AND TO INPUT
VIA PINS 4 THROUGH 7
        LDA A   #$0F
        STA A   $C001
NOW OUTPUT CONTROL CODES THAT SELECT I/O PORTS A AND B
        LDA A   #4
        STA A   $C002
        STA A   $C003
SET HAMMER PULSE, PW READY AND PW REL HIGH
SET START RIBBON MOTION LOW
        LDA A   #7
        STA A   $C001
LOAD $FF INTO UPPER BYTE OF SCRATCH READ/WRITE AREA
        CLR     SRCA
        COM     SCRA
PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O PORT B) FOR A 0 VALUE
START   LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$20        ISOLATE BIT 5
        BNE     START       IF NOT 0, RETURN TO START
```

INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF I/O PORT B. OUTPUT 1
TO BITS 2 AND 3 OF I/O PORT B

```
          LDA  A    #$C        LOAD MASK INTO ACCUMULATOR A
          STA  A    $C001      OUTPUT TO I/O PORT B
```

OUTPUT 0 TO BIT 3 OF I/O PORT B. THIS COMPLETES START RIBBON MOTION PULSE

```
          LDA  A    #4         LOAD MASK INTO ACCUMULATOR A
          STA  A    $C001      OUTPUT TO I/O PORT B
```

TEST FOR END OF PRINTWHEEL POSITIONING. BIT 5 OF I/O PORT B (FFI) WILL BE 1

```
LOP1      LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
          AND  A    #$20       ISOLATE BIT 5
          BEQ       LOP1       IF 0 RETURN TO LOP1
```

EXECUTE PRINTWHEEL SETTLING 2 MS DELAY

```
          LDX       #$FA       LOAD INITIAL TIME DELAY CONSTANT
LOP2      DEX                  DECREMENT INDEX REGISTER
          BNE       LOP2       RE-DECREMENT IF NOT ZERO
```

TEST PRINTHAMMER FIRING CONDITIONS

```
LOP3      LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
          ROL                  MOVE BIT 7 INTO CARRY
          BCC       PRD        IF CARRY IS ZERO BYPASS PRINTHAMMER FIRING
          AND  A    #$20       ISOLATE BIT 4 WHICH IS NOW BIT 5
          BEQ       LOP3       WAIT FOR NONZERO VALUE BEFORE FIRING
```

FIRE PRINTHAMMER

```
          LDA  A    $C001      SET HAMMER PULSE LOW. OUTPUT 0
          AND  A    #$FB       TO BIT 2 OF I/O PORT B
          STA  A    $C001
          LDA  A    $C000      INPUT ASCII CHARACTER TO ACCUMULATOR A
          AND  A    #$7F       MASK OUT HIGH ORDER BIT
```

COMPARE ASCII CODE WITH LOWEST LEGAL VALUE

```
          CMP  A    #20
          BLT       PRD        IF CODE IS $1F OR LESS, BYPASS HAMMER FIRING
```

COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE

```
          CMP  A    #$7A
          BGT       PRD        IF CODE IS $7B OR GREATER, BYPASS HAMMER FIRING
```

ASCII CODE IS VALID

```
          SUB  A    #$20       SUBTRACT $20
          STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
          LDX       SCRA
          LDA  A    INDEX,X    LOAD INDEX INTO ACCUMULATOR A
          ASL  A               MULTIPLY BY 2
          STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
          LDX       SCRA
          LDX       DELY,X     LOAD DELAY CONSTANT INTO INDEX REGISTER
LOP4      DEX                  EXECUTE LONG DELAY
          BNE       LOP4
          LDA  A    $C001      AT END OF DELAY OUTPUT 1 TO BIT 2
          ORA  A    #4         OF I/O PORT B. THIS SETS HAMMER PULSE HIGH
          STA  A    $C001
```

EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY

```
          LDX       #374       LOAD INITIAL TIME DELAY CONSTANT
LOP5      DEX                  EXECUTE LONG TIME DELAY       -
          BNE       LOP5
```

OUTPUT 1 TO BIT 0 OF I/O PORT B. THIS SETS PW REL HIGH

```
          LDA  A    $C001      INPUT I/O PORT B TO ACCUMULATOR A
          ORA  A    #1         SET BIT 0 TO 1
          STA  A    $C001      OUTPUT RESULT
```

```
        EXECUTE A 2 MS PRINTWHEEL READY DELAY
PRD     LDX     #$FA        LOAD INITIAL TIME DELAY
LOP6    DEX                 DECREMENT INDEX REGISTER
        BNE     LOP6        RE-DECREMENT IF NOT ZERO
        TEST FOR EOR DET (BIT 6 OF I/O PORT B) EQUAL TO 0 AS A
        PREREQUISITE FOR ENDING THE PRINT CYCLE
LOP7    LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$40        ISOLATE BIT 6
        BEQ     LOP7        RETURN AND RETEST IF 0
        AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT B TO 1
        THIS SETS PW RDY HIGH
        LDA A   $C001       INPUT I/O PORT B TO ACCUMULATOR A
        ORA A   #2          SET BIT 1 TO 1
        STA A   $C001       OUTPUT RESULT
        JMP     START       JUMP TO NEW PRINT CYCLE TEST
        INDEX TABLE FOLLOWS HERE
        ORG     $FF80
        Data representing 90 index entries follow here
        DELAYS TABLE FOLLOWS HERE
        ORG     $FFF0
        Data representing 6 delays follow here
```

Figure 4-6. A Simple Print Cycle Program

Chapter 5
A PROGRAMMER'S PERSPECTIVE

The program we developed in Chapter 4 is considerably shorter and easier to follow than the digital simulation of Chaptar 3. While we came a long way in Chepter 4 we still have a way to go. The program in Figure 4-6 treats the logic to be implemented as a single transfer function, but it is not a well written program.

To tha digital logic designer, one of the most confusing things about programming is the triviel eese with which you can do the same thing in ten different ways. Does this imply that some Implementations are more efficient than others? Indeed yes. To a great extent writing efficient programs is a talent, just as creating affi-cient digital logic is a talent; but thera ara certain rules which, if followed, will at leest help you avoid obvious mistakes. In this chapter wa ara going to take the program craated in Chapter 4 and look at it a little more cerefully.

SIMPLE PROGRAMMING EFFICIENCY

Tha first thing you should do, after writing a sourca program, is to go back over it, looking for elementary weys in which you can cut out instructions.

EFFICIENT TABLE LOOKUPS

On average, you will find that it is possible to reduce a program to two-thirds of its original length, simply by writing more efficient instruction sequences. In Figure 4-6, the most obvious example of sloppy programming involves the Index Table. The program loads a value between 1 and 6 from an Index Table byte, then multiplies this value by two before adding it to the base address of the Delay Table. **Why not directly store twice the index in the Index Table?** That cuts out one instruction as follows:
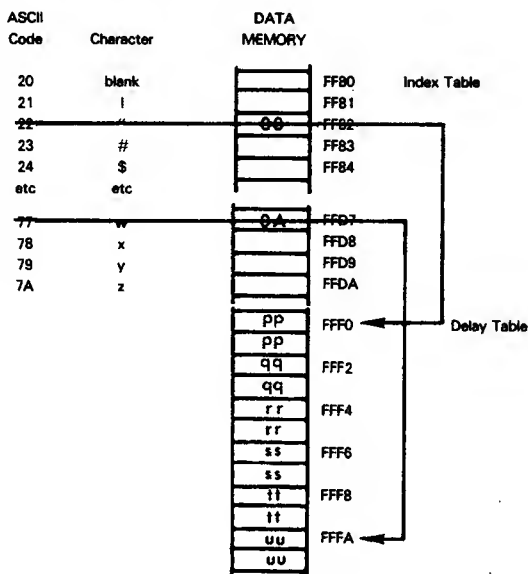
ASCII CODE IS VALID

```
        SUB  A    #$20       SUBTRACT $20
        STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX       SCRA
        LDA  A    INDEX,X    LOAD INDEX X2 INTO ACCUMULATOR A
        STA  A    SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX       SCRA
        LDX       DELY,X     LOAD DELAY CONSTANT INTO INDEX REGISTER
ASL
instruction
dropped
```

In the instruction sequence above, notice that one instruction has been removed following the shaded LDA instruction.

There are still a number of additional ways in which we can make the Delay Table lookup more efficient. **Why subtract 20₁₆ from the ASCII code,** for example: If we are going to add the ASCII code to a base address, there is nothing to stop us from equating the base address, represented by the symbol INDEX, to a value 20_{16} less than the first real Index Table byte. Our instruction sequence now collapses further, as follows:
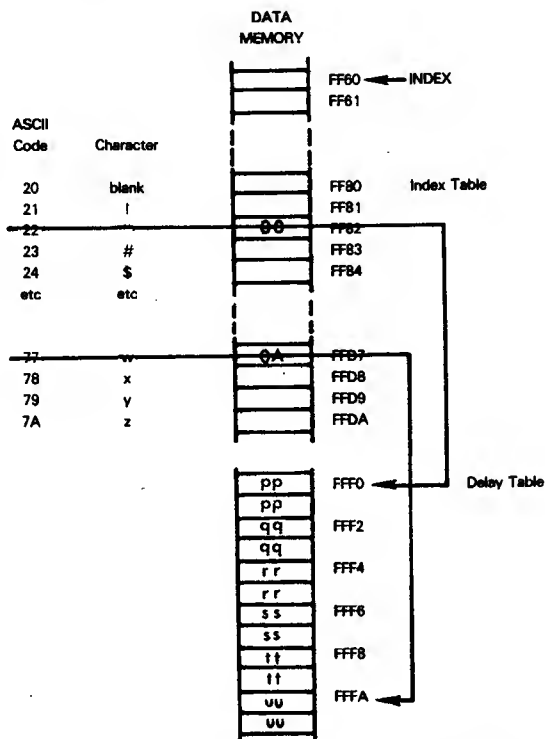

```
INDEX   EQU     $60 .       EQUATE INDEX TO TABLE BASE ADDRESS -$20
         -
         -
         -

COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CMP A   #$7A
        BGT     PRD         IF CODE IS 7B OR GREATER, BYPASS HAMMER FIRING
ASCII CODE IS INVALID
        STA A   SCRA + 1    MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX     SCRA
        LDA A   INDEX,X     LOAD INDEX X2 INTO ACCUMULATOR A
        STA A   SCRA + 1    MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
        LDX     SCRA
        LDX     DELY,X      LOAD DELAY CONSTANT INTO INDEX REGISTER
SUB
instruction
dropped
```

5-3

Okay, so INDEX is now being equated to 60_{16} — which means that we no longer need to subtract 20_{16} from the ASCII code. We have eliminated the SUB instruction which was above the shaded STA instruction.

Unfortunately there are no golden rules which, if followed, will ensure that you always write the shortest program possible. Once you have written a few programs, you will understand how individual instructions work; and that, in turn, generates efficiency. The purpose of the preceding pages has been to demonstrate the difference between a compact program and a straightforward program. If your product is to be produced in high volume, it behooves you to spend the time and money cutting down program size — then you may be able to eliminate some of your ROM chips.

HARDWARE UTILIZATION

All computer programmers try to write efficient assembly language programs. However, only microcomputer programmers must consider hardware utilization as an integral contributor to programming efficiency.

Now we used the MC6820 Peripheral Interface Adapter without handshaking or interrupts, accessing port bits in the most obvious way. Let us explore some of the alternatives.

HARDWARE-SPECIFIC INSTRUCTIONS

Observe that much of the time we are setting and resetting individual bits which become input and output signals.

Each I/O port of an MC6820 PIA has one control signal which can be configured as an output control. These are the CA2 and CB2 control signals. **You could generate two output signals using CA2 and CB2 as follows:**

MC6820 PIA
CONTROL
SIGNAL
OUTPUT

1) **First output the appropriate Control code to Control Registers A and B.** Here is the required Control code:

```
7 6 5 4 3 2 1 0 ◄── Bit No.
0 0 1 1 X 1 0 0   Control Register A or B

              Disable Interrupt Requests IRQA or IRQB
            Don't care since IRQA and IRQB disabled
          Select I/O port rather than Data Direction register
       Set CA2 or CB2 to X
     Read-only locations
```
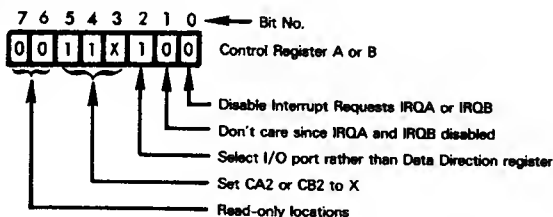
2) Subséquently **output $34** to the appropriate Control register **to reset CA2/CB2 low.** **Output $3C** to the appropriate Control register **to set CA2/CB2 high.**

Suppose PW READY is assigned to CB2 and PW REL is assigned to CA2. We will use Accumulator B to hold appropriate Control codes. These are the program modifications which result:

Figure 4-6 Program

New Program

```
                    -                                        -
                    -                                        -
                    -    .                                   -

NOW OUTPUT CONTROL CODES THAT          . NOW OUTRUT CONTROL CODES THAT
SELECT I/O PORTS A AND B                 SELECT I/O PORTS A AND B, AND
       LDA  A    #4         .            SET PW READY AND PW REL HIGH
       STA  A    $C002                          LDA  B    #$3C
       STA  A    $C003                          STA  B    $C002
SET HAMMER PULSE, PW READY AND                  STA  B    $C003
PW REL HIGH. SET START RIBBON                            -
MOTION LOW                                               -
       LDA  A    #7                                      -
       STA  A    $C001                                   -
                  -                                      -
                  -                                      -
                  -                                      -

INITIALIZE PRINT CYCLE. OUTPUT 0       INITIALIZE PRINT CYCLE. SET CA2
TO BITS 0 AND 1 OF I/O PORT B.         AND CB2 LOW.
OUTPUT 1 TO BITS 2 AND 3 of I/O                 LDA  B    #$34
PORT B                                          STA  B    $C002
       LDA  A    #$C                             STA  B    $C003
       STA  A    $C001                OUTPUT START RIBBON MOTION PULSE
OUTPUT 0 TO BIT 3 OF I/O PORT B.                LDA  A    #$C
THIS COMPLETES START RIBBON                     STA  A    $C001
MOTION PULSE                                     LDA  A    #$4
       LDA  A    #$4                             STA  A    $C001
       STA  A    $C001                                   -
                  -                                      -
                  -                                      -
                  -                                      -

OUTPUT 1 TO BIT 0 OF I/O PORT B.       SET CA2 HIGH. THIS SETS PW REL
THIS SETS PW REL HIGH                  HIGH
       LDA  A    $C001                          LDA  B    #$3C
       ORA  A    #1                             STA  B    $C002
       STA  A    $C001                                   -
                  -                                      -
                  -                                      -
                  -                                      -
                  -                                      -

AT END OF PRINT CYCLE SET BIT 1        SET CB2 HIGH. THIS SETS CH RDY
OF I/O PORT 2 TO 1. THIS SETS          HIGH
CH RDY HIGH                                     STA  B    $C003
       LDA  A    $C001                          JMP       START
       ORA  A    #2                                      -
       STA  A    $C001                                   -
       JMP       START                                  -
                  -                                      -
                  -                                      -
                  -                                      -
                  -                                      -
                  .
```

Let us look at the way in which program logic has changed when going from instructions as illustrated in Figure 4-6 to the new program.

In both cases three initial instructions are required to load appropriate Control codes into I/O Port A and B Control registers. In the new program however, PW READY and PW REL are simultaneously set high. This eliminates the subsequent need for two instructions to initialize I/O Port B output signals. In the program of Figure 4-6, it is necessary to initially set PW READY and PW REL high; simultaneously, we gratuitously set HAMMER PULSE high and START RIBBON MOTION low, even though these additional settings were not needed. Subsequent instructions initialize these additional signals appropriately. Thus, two instructions are saved in the new program.

The next point at which the new and old programs differ is when we initialize a print cycle by setting PW READY and PW REL low. In the old program, this simply required loading Accumulator A and outputting it to I/O Port B, which also set START RIBBON MOTION high. Two additional instructions are needed by the old program to reset START RIBBON MOTION low. In the new program, three instructions are required in order to set PW READY and PW REL low. The first instruction modified the Control code which we are maintaining in Accumulator B; two additional instructions output this Control code to its I/O Port A and B Control registers; these three instructions do nothing for START RIBBON MOTION, which is pulsed high by four instructions which follow.

In the new program, print cycle initialization has expanded from four instructions to seven instructions. Thus, the use of CA2 and CB2 has not proved economical in this instance. However, notice that the new program corresponds more closely to the timing diagram illustrated in Figure 3-2. This may be illustrated as follows:
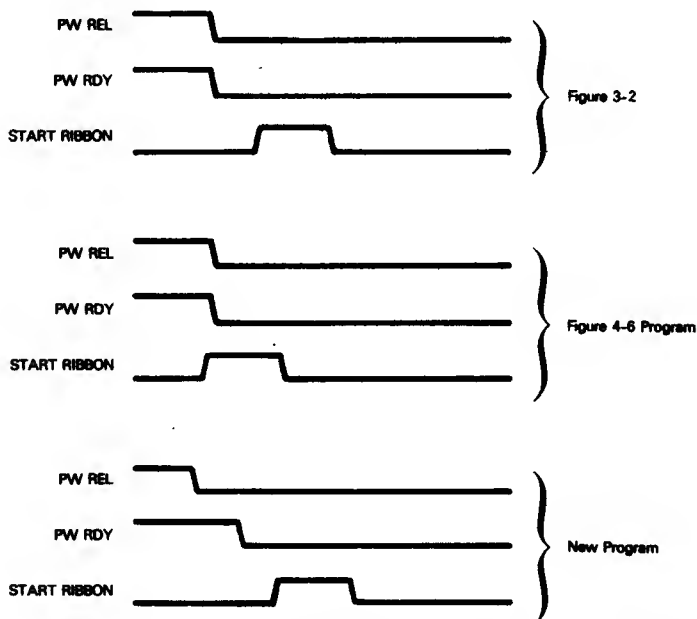
If some time delay is required between signals PW READY, PW REL and START RIBBON MOTION changing state, then the program illustrated in Figure 4-6 will be inadequate.

We do not manipulate PW READY or PW REL again until the end of the program. Here separate instruction sequences are required to set each of these two signals high, since a time interval separates the two signal changes of state. This being the case, the new program acquires a distinct advantage. The program illustrated in Figure 4-6 needs to input the contents of I/O Port B, set a single bit to 1, then output this modified data back to I/O Port B. These three instructions need to be executed twice, once for each signal's change of state. In the new program we load a new Control code into Accumulator B, then output Accumulator B contents to each I/O port Control register. This sequence demonstrates the economy of using control signals CA2 and CB2, rather than modifying individual bits of I/O ports.

DIRECT USE OF HARDWARE FEATURES

We ere going to essume that external logic uses the PRINTWHEEL READY signal to ensure thet it does not attempt to Input e new character code until the prior cherecter code has been processed. We expend some instructions setting PRINT- WHEEL READY low et the beginning of the print cycle, then resetting it high et the end of the print cycle.

Without e clear definition of the logic external to our microcomputer system, we have no wey of knowing whethar the PRINTWHEEL RELEASE and PRINTWHEEL READY output signels ere needed externelly to define specific time deleys, or whethar they are simply being used to ensure that wa allow one character to complete printing befora trying to start printing tha next charactar. if the only function of the PRINTWHEEL READY signal is to ensure that e new cherecter is not input to the microcomputer system before the old cherecter hes been printed, then we cen dispense with the PRINTWHEEL READY signal end rapleca it with eutomatic input hendsheking logic.

Input interrupt or programmed handshaking applies to I/O Port A only. I/O Port A is being used by external logic to input the ASCII character code which is to be printed; our MC6820 PIA is therefore correctly configured to use input handshaking. Input handshaking may be illustrated as follows:

MC6820
INPUT
HANDSHAK-
ING

E

R/W

D0-D7   Reed Deta
         from I/O Port A

CA2

CA1

CA2 is output low es soon es the
ASCII code is reed out of I/O Port
A end into Accumulator A

External logic must change the
level of CA1 when placing a new
ASCII code at I/O Port A; this
causes CA2 to go high again
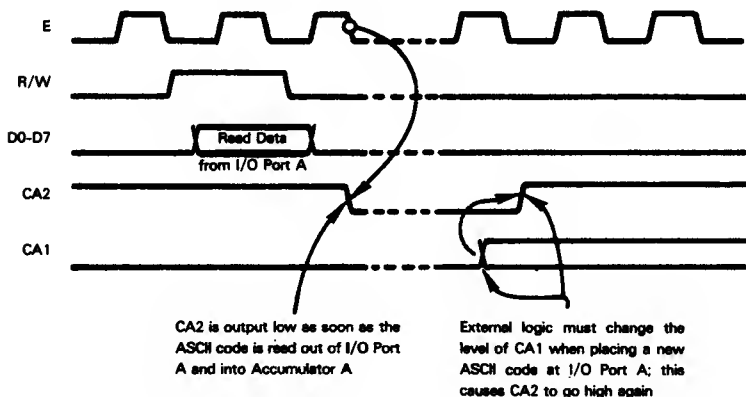
CA2 is output low on the trailing edge of the synchronization signal E, after the CPU has read the contents of I/O Port A. Thus, as soon as we read the ASCII character from I/O Port A, CA2 will be output low. External logic may use a low CA2 signal level as an indicator that another ASCII character must be written to I/O Port A. When external logic writes another character to I/O Port A, it must simultaneously input an active pulse via CA1; what constitutes an active pulse is determined by the Control code written to the I/O Port A Control register. We will assume that external logic must input CA1 high when it writes new data to I/O Port A.

We are going to disable interrupts associated with active transitions of CA1 since it is possible for external logic to write a new character to I/O Port A before the print cycle for the previous character has been executed. It would be disastrous if a program interrupt occurred while we were still executing the balance of the print cycle required to print the character which had just been read from I/O Port A. Instead of using the interrupt which can be generated by an active transition of CA1, our program logic will test the status of Control Register A, bit 7, in order to determine whether the active transition of CA1 has occurred. Recall that an interrupt request generated by an active transition of CA1 will also set bit 7 of the I/O Port A Control register; by polling this bit in between print cycles we can determine when to initiate a new print cycle.

Here is the Control code that must now be written to Control Register A in order to enable input handshaking:
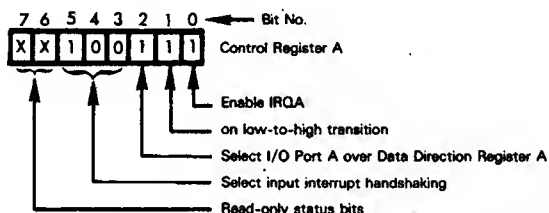
Observe that while using input handshaking you cannot use CA2 as an independent output control signal, as we just discussed before looking at the direct use of hardware features.

Here are the program changes which must be made upon going to input handshaking:

Figure 4-6 Program	New Program

```
                    -                                    -
                    -                                    -
                    -                                    -
NOW OUTPUT CONTROL CODES THAT            NOW OUTPUT CONTROL CODES THAT
SELECT I/O PORTS A AND B                 SELECT I/O PORT A WITH PROGRAMMED
        LDA  A    #4                     HANDSHAKING AND I/O PORT B WITH
        STA  A    $C002                  SIMPLE DATA I/O
        STA  A    $C003                          LDA  A    #$27
                    -                            STA  A    $C002
                    -                            LDA  A    #4
                    -                            STA  A    $C003
                    -                                    -
                    -                                    -
                    -                                    -
PRINT CYCLE PROGRAM                      PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI         IN BETWEEN PRINT CYCLES TEST THE CA1
(BIT 5 OF I/O PORT B) FOR A 0            STATUS (BIT 7 OF CONTROL REGISTER
VALUE                                    A) TO SEE IF CA1 HAS MADE AN ACTIVE
START   LDA  A    $C001                  TRANSITION
        AND  A    #$20                   START   LDA  A    $C002
        BNE       START                          ROL
                                                 BCC       START
INITIALIZE PRINT CYCLE.                  INITIALIZE PRINT CYCLE
OUTPUT 0 TO BITS 0 AND 1 OF I/O          OUTPUT 0 TO BIT 0 OF I/O PORT B
PORT B. OUTPUT 1 TO BITS 2 AND 3         OUTPUT 1 TO BITS 2 AND 3 OF I/O
OF I/O PORT B                            PORT B
        LDA  A    #$C                            LDA  A    #$C
        STA  A    $C001                          STA  A    $C001
                    -                                    -
                    -                                    -
                    -                                    -
AT END OF PRINT CYCLE SET BIT 1          AT END OF PRINT CYCLE RETURN TO
OF I/O PORT B TO 1. THIS SETS PW         START
RDY HIGH                                         JMP       START
        LDA  A    $C001
        ORA  A    #2                        •
        STA  A    $C001
        JMP       START
```

The fact that we are testing bit 7 of Control Register A in order to start a new print cycle also means that the PW READY signal disappears. We do not save any instructions at the beginning of the program since we still have to initialize other signals. At the end of the program, however, there are three instructions which in the program of Figure 4-6 simply set PW READY high. We may eliminate these three instructions in the new program, since PW READY no longer exists.

SUBROUTINES

If you look again at the program in Figure 4-6, you will notice that at two points within this program we execute identical instruction sequences to create a 2 millisecond delay. Now it takes only three instructions to execute a 2 millisecond delay, so the fact that these three instructions have been repeated is no big tragedy. If you think about it, however, the potential exists for some very uneconomical memory utilization in longer programs.

We have kept our program simple in Chapter 4 because it must remain small enough to handle in a book; but project, if you will, a more complex routine where a 30-instruction sequence needs to be repeated, rather than a three-instruction sequence. We must now find some way of including the instruction sequence just once, then branching to this single sequence from a number of different locations within a program, as needed. That is what a subroutine will do for you.

Let us take the three instructions which execute a 2 millisecond delay and convert them into a subroutine. This is what happens to relevant portions of the program:

```
                ORG        $FC00
        SYSTEM RESET AND INITIALIZATION
        FIRST INITIALIZE THE STACK POINTER
                LDS        #$007F
        OUTPUT CONTROL CODE TO I/O PORT A CONTROL REGISTER
                -
                -
                -
        EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
                JSR        D2MS
                -
                -
                -
        EXECUTE A 2 MS PRINTWHEEL READY DELAY
        PRD     JSR        D2MS
                -
                -
                -
                JMP        START       JUMP TO NEW PRINT CYCLE TEST
        SUBROUTINE TO EXECUTE A 2 MS DELAY
        D2MS    LDX        #$FA        LOAD INITIAL TIME DELAY
        LOPD    DEX                    DECREMENT INDEX REGISTER
                BNE        LOPD        REDECREMENT IF NOT ZERO
                RTS                    RETURN FROM SUBROUTINE
        INDEX TABLE FOLLOWS HERE
                -
                -
                -
```

In order to understand how a subroutine works, we will assign some arbitrary memory addresses for our source program's object code; we will show, step-by-step, what happens when a subroutine is called and what happens upon returning from the subroutine. First of all, here is the assumed memory map, which conforms to Figure 4-2:

PROGRAM
MEMORY

	LDS	#$007F	8E	FC00
	-		00	FC01
	-		7F	FC02
	-			
	-			
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
	AND A	#$20	84	FC1F
			20	FC20
	BEQ	LOP1	27	FC21
			F9	FC22
	JSR	D2MS	BD	FC23
			FC	FC24
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
	-		C0	FC27
	-		01	FC28
	-			
	JMP	START	7E	FCF0
			FC	FCF1
			0C	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
	BNE	LOPD	26	FCF7
			FD	FCF8
	RTS		39	FCF9

Registers:
A
B
IX
SP
PC
I

DATA
MEMORY

	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
	007E
	007F

SUBROUTINE CALL

Suppose we are about to execute the first JSR D2MS instruction. At this point registers will contain the following data:

PROGRAM
MEMORY

	LDS	#$007F	8E	FC00
	-		00	FC01
	-		7F	FC02
	-			
	-			
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
	AND A	#$20	84	FC1F
			20	FC20
	BEQ	LOP1	27	FC21
			F9	FC22
	JSR	D2MS	BD	FC23
			FC	FC24.
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
	-		C0	FC27
	-		01	FC28
	-			
	JMP	START	7E	FCF0
			FC	FCF1
			0C	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
	BNE	LOPD	26	FCF7
			FD	FCF8
	RTS		39	FCF9

A	20
B	
IX	
SP	007F
PC	FC23
I	27

DATA
MEMORY

	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
	007E
	007F

The Program Counter (PC) addresses the first byte of the Jump-to-Subroutine (JSR) instruction's object code; this address is FC23₁₆. The Instruction register holds the object code for the most recently executed instruction; this is a BEQ instruction located at byte FC21₁₆. The Stack Pointer, you will notice, was initialized at the beginning of the program; it contains 007F₁₆. According to Figure 4-2, this is the address of the first byte of read/write memory. Since the stack has not been used, the Stack Pointer will still contain 007F₁₆.

The Accumulator contains 20_{16} because this was the condition which caused execution to break out of the holding loop starting at LOP1.

Now when the JSR instruction is executed, steps occur as follows:

The JSR instruction object code is loaded into the Instruction register and the Program Counter is incremented:

			PROGRAM MEMORY	
LDS	#$007F		8E	FC00
-			00	FC01
-			7F	FC02
-				
-				
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
AND A	#$20		84	FC1F
			20	FC20
BEQ	LOP1		27	FC21
			F9	FC22
JSR	D2MS		BD	FC23
			FC	FC24
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
			C0	FC27
			01	FC28
-				
-				
JMP	START		7E	FCF0
			FC	FCF1
			0C	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
BNE	LOPD		26	FCF7
			FD	FCF8
RTS			39	FCF9

Registers:

A	20
B	-
IX	
SP	007F
PC	FC24
I	BD

DATA MEMORY

	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
	007E
	007F
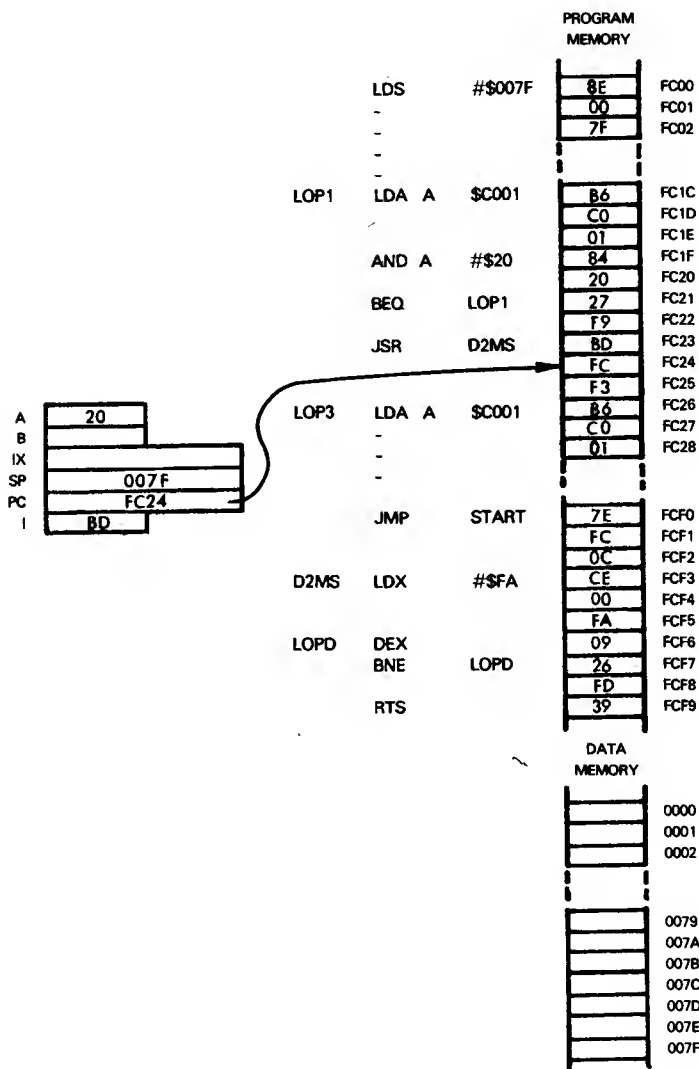
The Program Counter is incremented by 2 to bypass the JSR address. This incremented value is saved in the first two stack bytes. The JSR address is then loaded into the Program Counter. The Stack Pointer is decremented by 2 so that it addresses the first free stack byte:

			PROGRAM MEMORY	
LDS		#$007F	8E	FC00
–			00	FC01
–			7F	FC02
–				
–				
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
	AND A	#$20	84	FC1F
			20	FC20
	BEQ	LOP1	27	FC21
			F9	FC22
	JSR	D2MS	BD	FC23
			FC	FC24
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
			C0	FC27
–			01	FC28
–				
–				
	JMP	START	7E	FCF0
			FC	FCF1
			0C	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
	BNE	LOPD	26	FCF7
			FD	FCF8
	RTS		39	FCF9

Registers:
A 20
B
IX
SP 007D
PC
I BD

DATA MEMORY	
	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
FC	007E
26	007F

The next instruction executed has its object code stored in memory byte FCF3$_{16}$; this is the
memory byte now addressed by the Program Counter:

PROGRAM
MEMORY

LDS		#$007F	8E	FC00
-			00	FC01
-			7F	FC02
-				
-				
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
	AND A	#$20	84	FC1F
			20	FC20
	BEQ	LOP1	27	FC21
			F9	FC22
	JSR	D2MS	BD	FC23
			FC	FC24
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
			C0	FC27
			01	FC28
	-			
	-			
	-			
	JMP	START	7E	FCF0
			FC	FCF1
			06	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
	BNE	LOPD	26	FCF7
			FD	FCF8
	RTS		39	FCF9

A   20
B
IX
SP   007D
PC   FCF3
I   CE

DATA
MEMORY

	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
FC	007E
26	007F

Instructions within the 2 millisecond delay loop are now executed repetitively until the Index
register contents decrement from 01 to 00.

SUBROUTINE RETURN

When the Index register finally decrements from 01 to 00, execution passes to the Return-from-Subroutine (RTS) instruction. This instruction increments the contents of the Stack Pointer by 2, then moves the contents of the two top stack bytes into the Program Counter. Thus, program execution returns to the instruction that follows the Jump-to-Subroutine (JSR).

PROGRAM MEMORY

LDS		#$007F	8E	FC00
-			00	FC01
-			7F	FC02
-				
-				
LOP1	LDA A	$C001	B6	FC1C
			C0	FC1D
			01	FC1E
AND A		#$20	84	FC1F
			20	FC20
BEQ		LOP1	27	FC21
			F9	FC22
JSR		D2MS	BD	FC23
			FC	FC24
			F3	FC25
LOP3	LDA A	$C001	B6	FC26
-			C0	FC27
-			01	FC28
-				
JMP		START	7E	FCF0
			FC	FCF1
			06	FCF2
D2MS	LDX	#$FA	CE	FCF3
			00	FCF4
			FA	FCF5
LOPD	DEX		09	FCF6
BNE		LOPD	26	FCF7
			FD	FCF8
RTS			39	FCF9

Registers:

A	00
B	
IX	
SP	007F
PC	FC26
I	39

DATA MEMORY

	0000
	0001
	0002
	0079
	007A
	007B
	007C
	007D
FC	007E
26	007F

In summary, this is what happened:

When the Jump-to-Subroutine instruction was executed, the address of the next instruction was saved in the stack. The Jump-to-Subroutine instruction provided the address of the next instruction to be executed.

The next instruction to be executed was the first instruction of the subroutine.

The last instruction of the subroutine merely caused the address saved at the top of the stack to be returned to the Program Counter; and this, in turn, caused execution to branch back to the instruction following the Jump-to-Subroutine.

WHEN TO USE SUBROUTINES

There is a price associated with using subroutines:

1) **Each JSR instruction represents three additional bytes of object code.**

2) **The instruction sequence which has been moved to the subroutine must have an appended Return instruction which costs one byte of object code.**

Let us first look at our specific case. The three instructions which constitute the 2 millisecond delay occupy 6 bytes of object code. These three instructions occur twice; therefore, combined, they occupy 12 bytes of object code. When moved to a subroutine, adding the Return instruction increases the object code bytes from 6 to 7. In addition, there are two JSR instructions and each requires 3 bytes of object code — which means that the two instructions, plus the subroutine, generate 13 bytes of object code. This may be illustrated as follows:

OLD PROGRAM

	LDX	#$FA	CE
			00
			FA
LOP2	DEX		09
	BNE	LOP2	26
			FD

PDR	LDX	#$FA	CE
			00
			FA
LOP6	DEX		09
	BNE	LOP6	26
			FD

NEW PROGRAM

	JSR	D2MS	BD
			FC
			F3
	JSR	D2MS	BD
			FC
			F3

D2MS	LDX	#$FA	CE
			00
			FA
LOPD	DEX		09
	BNE	LOPD	26
			FD
	RTS		39

In our specific case, therefore, moving the 2 millisecond delay instruction sequence into a subroutine has cost us one byte of object code.

Now these comments do not imply that subroutines are a dubious programming feature, to be used sparingly; on the contrary, it is hard to conceive of any program which, when well written, will not include some subroutines. But bear in mind that **there is a minimum subroutine size below which subroutines in general become uneconomical.**

Suppose there are N bytes of object code in an instruction sequence which you are planning to convert into a subroutine.

Suppose the **N bytes of object code occur M times;** that means when the N bytes of object code become a subroutine, it will be called by M JSR instructions.

Without subroutines, M x N bytes will be consumed repeating N bytes M times. With subroutines, the number of bytes consumed is:

$$3M + (N + 1) + 3 + 2$$

- 2 bytes on stack for address storage
- Bytes for stack initialization instruction object codes
- Subroutine, including RET instruction
- M subroutine calls, assuming extended direct addressing
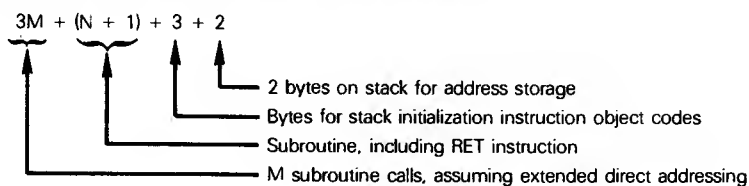
For the subroutine to be worthwhile, 3M+N+6 must be less than M x N.

Table 5-1 shows the minimum economic subroutine length as a function of the number of subroutine calls.

Table 5-1. The Shortest Economic Subroutine Length As A Function
Of The Number Of Times The Subroutine Is Called

Number Of Subroutine Calls (M)	Minimum Economic Subroutine Length (N)
2	12 Bytes
3	8 Bytes
4	6 Bytes
5	6 Bytes
10	4 Bytes
20	4 Bytes

MULTIPLE SUBROUTINE RETURNS

Even though none of the repeated instruction sequences within the program in Figure 4-6 are long enough to justify being turned into a subroutine, we will nonetheless explore the potential of subroutines further.

Consider the printhammer firing instruction sequence in Figure 4-6. Given the program as illustrated, this instruction sequence occurs just once, which means that converting it into a subroutine would make no sense. **It is possible to imagine a more extensive program which performs a wide variety of printer interface operations, such that printhammer firing logic might be triggered for a number of different reasons. Since the printhammer firing logic consists of a fairly long set of instructions, putting these instructions in a subroutine would be absolutaly mandatory. Consider the following subroutine implementation:**

```
PRINTHAMMER FIRING SUBROUTINE
PFIR     LDA  A   $C001      INPUT I/O PORT B TO ACCUMULATOR A
         ROL                 MOVE BIT 7 INTO CARRY
         BCC      RTRN       IF CARRY IS ZERO RETURN FROM SUBROUTINE
         AND  A   #$20       ISOLATE BIT 4 WHICH IS NOW BIT 5
         BEQ      RTRN       IF ZERO, RETURN FROM SUBROUTINE
FIRE PRINTHAMMER
         LDA  A   $C001      SET HAMMER PULSE LOW, OUTPUT 0
         AND  A   #$FB       TO BIT 2 OF I/O PORT B
         STA  A   $C001
         LDA  A   $C000      INPUT ASCII CHARACTER TO ACCUMULATOR A
         AND  A   #$7F       MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
         CMP  A   #$20
         BLT      RTRN       IF CODE IS $1F OR LESS, RETURN FROM SUBROUTINE
COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
         CMP  A   #$7A
         BGT      RTRN       IF CODE IS $7B OR GREATER, RETURN FROM SUBROUTINE
ASCII CODE IS VALID
         STA  A   SCRA + 1   MOVE ACCUMULATOR A  CONTENTS TO INDEX REGISTER
         LDX      SCRA
         LDA  A   INDX,X     LOAD INDEX INTO ACCUMULATOR A
         STA  A   SCRA + 1   MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
         LDX      SCRA
         LDX      DELY,X     LOAD DELAY CONSTANT INTO INDEX REGISTER
         JSR      LDLY       EXECUTE LONG DELAY
         LDA  A   $C001      AT END OF DELAY OUTPUT 1 TO BIT 2
         ORA  A   #4         OF I/O PORT B. THIS SETS HAMMER PULSE HIGH
         STA  A   $C001
EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
         LDX      #374       LOAD INITIAL TIME DELAY CONSTANT
         JSR      LDLY       EXECUTE LONG TIME DELAY
OUTPUT 1 TO BIT 0 OF I/O PORT B. THIS SETS PW REL HIGH
         LDA  A   $C001      INPUT I/O PORT B TO ACCUMULATOR A
         ORA  A   #1         SET BIT 0 TO 1
         STA  A   $C001      OUTPUT RESULT
RTRN     RTS                 RETURN FROM SUBROUTINE
```

The subroutine illustrated above fires the printhammer only if all necessary conditions have been met; a quick axit is executed if any firing condition has not been met.

We have added e subroutine within the subroutine. The long delay instruction sequence hes been moved to e subroutine, the first instruction of which is labeled LDLY. This is referred to es e "nested subroutine".

NESTED SUBROUTINES

One novel feature of subroutine LDLY is that it requires the initial delay constant to be stored in the Index register. **The initiel deley constant becomes e peremeter,** which allows one subroutine to implement a complete spectrum of time delays. Subroutine parameters are a very important feature of subroutine use.

SUBROUTINE PARAMETER

Subroutine PFIR is not es useful es it could be. There ere four conditional returns from this subroutine, eech of which is triggered by a different invalid condition. There is elso a subroutine return following velid printhemmer firing.

How is the calling progrem to know whether the printhemmer wes or wes not fired after PFIR was called? Testing statuses is not very safe, since we cannot be certain what happens to status conditions during execution of the printhammer firing instructions themselves. You cannot test the Carry status to determine whether the BCC instruction caused an exit from subroutine PFIR; this is because you cannot tell what happens to the Carry status while the rest of the subroutine executes. For example, the Carry status will be modified by execution of the CMP instructions.

Subroutines which contain e large number of conditionel error exits, in addition to e standerd return, wiJJ often contain logic which returns to e number of different instructions in the calling program. Take the cese of subroutine PFIR. The instruction sequence whlch calls this subroutine may eppeer es follows:

```
              -
              -
              -
RT0    JSR    PFIR      CALL PRINTHAMMER FIRING SUBROUTINE
       JMP    RT1       RETURN HERE FOR PRINTWHEEL REPOSITIONING
       JMP    RT0       RETURN HERE FOR HAMMER INTERLOCK LOW
       JMP    RT2       RETURN HERE FOR ASCII CODE LESS THAN $20
       JMP    RT3       RETURN HERE FOR ASCII CODE GREATER THAN $7A
INSTRUCTIONS WHICH FOLLOW ARE EXECUTED AFTER VALID PRINTHAMMER FIRING
              -
              -
              -

INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR PRINTWHEEL REPOSITIONING
RT1    -
              -
              -
INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR ASCII CODE LESS THAN $20
RT2    -
              --
              -
INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR ASCII CODE GREATER THAN $7A
RT3    -
              -
              -
```

Now for this scheme to work, subroutine PFIR must increment the return address, which is stored in the top two bytes of the stack, every time a conditional return is executed. Subroutine PFIR is therefore modified as follows:

```
PRINTHAMMER FIRING SUBROUTINE
PFIR     LDA  A    $C001         INPUT I/O PORT B TO ACCUMULATOR A
         ROL                     MOVE BIT 7 INTO CARRY
         BCC       RTRN + 9      IF CARRY IS ZERO, RETURN FROM SUBROUTINE
         AND  A    #$20          ISOLATE BIT 4 WHICH IS NOW BIT 5
         BEQ       RTRN + 6      IF ZERO, RETURN FROM SUBROUTINE
FIRE PRINTHAMMER
         LDA  A    $C001         SET HAMMER PULSE LOW, OUTPUT 0
         AND  A    #$FB          TO BIT 2 OF I/O PORT B
         STA  A    $C001
         LDA  A    $C000         INPUT ASCII CHARACTER TO ACCUMULATOR A
         AND  A    #$7F          MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
         CMP  A    #$20
         BLT       RTRN + 3      IF CODE IS $1F OR LESS, RETURN FROM SUBROUTINE
COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
         CMP  A    #$7A
         BGT       RTRN          IF CODE IS $7B OR GREATER, RETURN FROM SUBROUTINE
ASCII CODE IS VALID
         STA  A    SCRA + 1      MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
         LDX       SCRA
         LDA  A    INDX,X        LOAD INDEX INTO ACCUMULATOR A
         STA  A    SRCA + 1      MOVE ACCUMULATOR A CONTENTS TO INDEX REGISTER
         LDX       SCRA
         LDX       DELY,X        LOAD DELAY CONSTANT INTO INDEX REGISTER
         JSR       LDLY          EXECUTE LONG DELAY
         LDA  A    $C001         AT END OF DELAY OUTPUT 1 TO BIT 2
         ORA  A    #4            OF I/O PORT B. THIS SETS HAMMER PULSE HIGH
         STA  A    $C001
EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
         LDX       #374          LOAD INITIAL TIME DELAY CONSTANT
         JSR       LDLY          EXECUTE LONG DELAY
OUTPUT 1 TO BIT 0 OF I/O PORT B. THIS SETS PW REL HIGH
         LDA  A    $C001         INPUT I/O PORT B TO ACCUMULATOR A
         ORA  A    #1            SET BIT 0 TO 1
         STA  A    $C001         OUTPUT RESULT
         JMP       RTRN + 12     VALID RETURN
RTRN     JSR       INCR          JUMP HERE FOR ASCII CODE GREATER THAN $7A
         JSR       INCR          JUMP HERE FOR ASCII CODE LESS THAN $20
         JSR       INCR          JUMP HERE FOR HAMMER INTERLOCK LOW
         JSR       INCR          JUMP HERE FOR PRINTWHEEL REPOSITIONING
         RTS                     JUMP HERE FOR VALID RETURN
```

SUBROUTINE TO INCREMENT TOP TWO STACK BYTES FOLLOWS

```
INCR    INS                 INCREMENT STACK POINTER TWICE TO
        INS                 BYPASS INCR RETURN ADDRESS
        TSX                 MOVE STACK POINTER PLUS 1 TO INDEX REGISTER
        LDX     0,X         LOAD RETURN ADDRESS INTO INDEX REGISTER
        INX                 INCREMENT BY 3 TO BYPASS ONE JUMP
        INX                 INSTRUCTION FOLLOWING SUBROUTINE CALL
        INX
        STX     SCRA        SAVE INCREMENTED ADDRESS IN SCRATCH MEMORY
        LDA  A  SCRA        LOAD INCREMENTED ADDRESS INTO A (HIGH
        LDA  B  SCRA + 1    ORDER BYTE) AND B (LOW ORDER BYTE)
        INS                 INCREMENT STACK POINTER
        INS
        PSH  B              PUSH RETURN ADDRESS BACK ONTO STACK
        PSH  A
        DES                 DECREMENT STACK POINTER TWICE
        DES
        RTS                 RETURN FROM SUBROUTINE
```

**Subroutine INCR is interesting; it shows how the stack
may be manipulated. Let us take a look at what happens.**

**STACK
MANIPULATION**

As soon as subroutine INCR is entered, the Stack Pointer contents are
increased by two. This has the effect of addressing the PFIR return address rather than the INCR
return address:
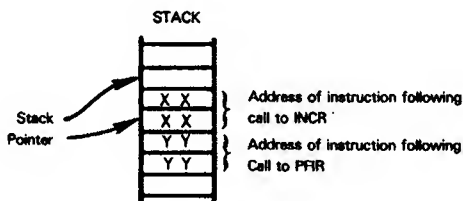
The next two instructions load the PFIR return address into the Index register. This transfer is quite
simple to execute. The TSX instruction moves the contents of the Stack Pointer into the Index
register, then increments the contents of the Index register. Using an LDX instruction with index-
ed, direct addressing, we can now load the PFIR address back into the Index register. This may be
illustrated as follows:
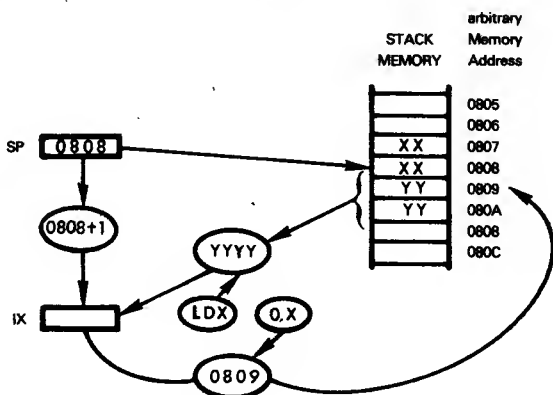
With the PFIR return address in the Index register, we can simply increment the Index register contents three times; we must increase the Index register contents by three since, if you look at the call to PFIR, it is followed by three Jump instructions; each Jump instruction is three bytes long. Every time subroutine INCR is called, its purpose is to increase the return address value by three, thus causing the return to strike a Jump instruction that is one further removed from the call. This may be illustrated as follows:

```
RT0     JSR     PFIR
        JMP     RT1 ◄──────────── RETURN HERE AFTER CALLING INCR ONCE
        JMP     RT2 ◄──────────── RETURN HERE AFTER CALLING INCR TWICE
        JMP     RT3 ◄──────────── RETURN HERE AFTER CALLING INCR THREE TIMES
        ? ◄──────────────────── RETURN HERE AFTER CALLING INCR FOUR TIMES
```

Unfortunately, getting the incremented return address back into the stack is not simple. We cannot store the Index register contents using direct, indexed addressing, since the Index register now contains the data which must be written. The Index register therefore cannot be used simultaneously to hold address data. In consequence, we store the Index register contents in two bytes of read/write memory, then load the data into the A and B Accumulators. Next we increment the Stack Pointer to address the low order byte of the PFIR return address space. And at last we can push the incremented address back onto the stack.

Finally we can decrement the Stack Pointer contents twice, so that the Stack Pointer is accessing the INCR return address. We are now ready to return from subroutine INCR.

MACROS

When talking about subroutines, we glossed over one consideration — you, the programmer. Subroutines have an additional value, in that if they reduce the number of source program instructions, then they will also reduce the amount of time you spend writing the source program, since program writing time will be directly proportional to program length.

Let us take another look at the 2 millisecond time delay subroutine:

```
           Old Program                      New Program
           LDX     #$FA                      JSR     D2MS
LOP2       DEX                                 ͞
           BNE     LOP2                      JSR     D2MS
             ͞                                  ͞
PDR        LDX     #$FA            D2MS      LDX     #$FA
LOP6       DEX                     LOPD      DEX
           BNE     LOP6                      BNE     LOPD
             ͞                                RTS
```

Given just two calls to subroutine D2MS, as occurs in the program of Figure 4-6, the old and new instruction sequences illustrated above include exactly the same number of source program instructions — six. But the old program requires 12 bytes of object code whereas the new program requires 13 bytes of object code.

What happens when there are three calls to D2MS? This may be illustrated as follows:

```
           Old Program                      New Program
           LDX     #$FA                      JSR     D2MS
LOP2       DEX                                 ͞
           BNE     LOP2                      JSR     D2MS
             ͞                                  ͞
           LDX     #$FA                      JSR     D2MS
LOP3       DEX                                 ͞
           BNE     LOP3            D2MS      LDX     #$FA
             ͞                       LOPD      DEX
           LDX     #$FA                      BNE     LOPD
```

5-23

```
        LOP4    DEX                              RTS
                BNE     LOP4
            9 instructions                   7 instructions
            18 bytes                         16 bytes
```

**Subroutines can decrease the length of your source program, while increasing the
length of your object program, and the program's execution time; or subroutines
can decrease source and object program lengths.**

**Macros decrease the length of your source program, but have absolutely no effect
on your object program.**

WHAT IS A MACRO?

**A Macro is a form of programming "shorthand"; it allows you to define an instruc-
tion sequence with a single mnemonic.**

Consider the 2 millisecond time delay instruction sequence; we can
define it as a macro, labeled D2MS, as follows:

**MACRO
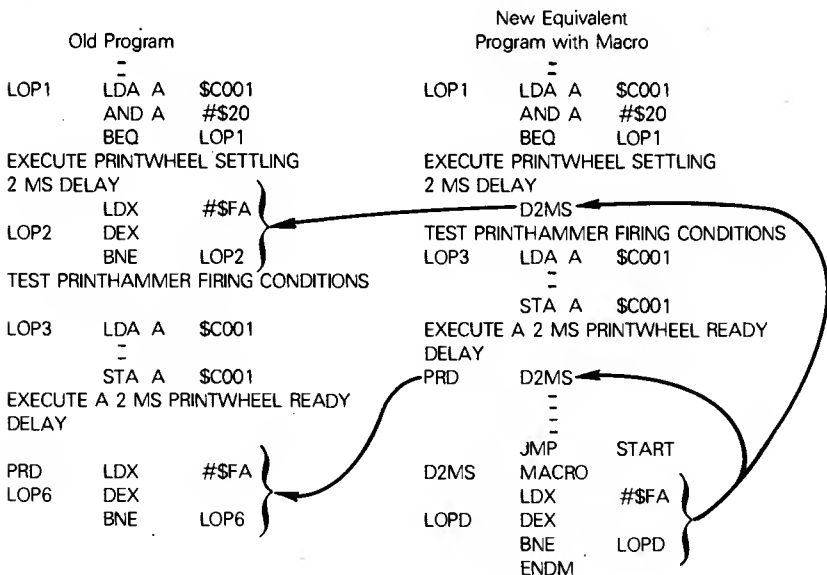DEFINITION**

```
        D2MS    MACRO
                LDX     #$FA
        LOPD    DEX
                BNE     LOPD
                ENDM
```

The two shaded instructions above are assembler directives; they bracket
a sequence of instructions which henceforth can be identified, as a group,
using the label of the MACRO assembler directive.

**MACRO
ASSEMBLER
DIRECTIVES**

This is how we would use the 2 millisecond time delay in our print cycle
program:

```
                                              New Equivalent
        Old Program                        Program with Macro

LOP1    LDA  A  $C001              LOP1    LDA  A  $C001
        AND  A  #$20                       AND  A  #$20
        BEQ     LOP1                        BEQ     LOP1
EXECUTE PRINTWHEEL SETTLING        EXECUTE PRINTWHEEL SETTLING
2 MS DELAY                         2 MS DELAY
        LDX     #$FA                                D2MS
LOP2    DEX                        TEST PRINTHAMMER FIRING CONDITIONS
        BNE     LOP2               LOP3    LDA  A  $C001
TEST PRINTHAMMER FIRING CONDITIONS
                                           STA  A  $C001
LOP3    LDA  A  $C001              EXECUTE A 2 MS PRINTWHEEL READY
                                   DELAY
        STA  A  $C001              PRD     D2MS
EXECUTE A 2 MS PRINTWHEEL READY
DELAY                                      JMP     START
PRD     LDX     #$FA               D2MS    MACRO
LOP6    DEX                                LDX     #$FA
        BNE     LOP6               LOPD    DEX
                                           BNE     LOPD
                                           ENDM
```

When the Assembler encounters the symbol D2MS in the mnemonic field, what it does is replace this symbol with the instructions bracketed by directives MACRO and ENDM. The Assembler knows which macro to use in the event that your program has more than one macro, since the symbol in the mnemonic field must be identical to the label of a MACRO directive.

Notice that the Assembler can also do a certain amount of housekeeping associated with the use of macros. The "Old Program" illustrated above has labels LOP2 and LOP6 for the two DCR instructions. The "New Program" has a single label, LOPD, within the macro. The Assembler is smart enough to know that a label appearing within a macro definition must become a series of separate labels when the macro subsequently is inserted a number of times into the source program.

To summarize, you simply take a sequence of repeated instructions, bracket them with MACRO and ENDM directives, then give the macro directive a unique label. Now use the MACRO's label as though it were an instruction mnemonic. The macro definition must appear once and only once, somewhere in the source program. It is a good idea to

> MACRO
> DEFINITION
> LOCATION
> IN A SOURCE
> PROGRAM

collect all of your macros and insert them at the beginning, or at the end of the entire source program.

MACROS WITH PARAMETERS

Instructions within a macro can have variable operands; for example, we can create a variable time delay macro as follows:

```
DVMS    MACRO    TIME
        LDX      #TIME
LOPD    DEX
        BNE      LOPD
        ENDM
```

Symbols appearing in the MACRO directive's operand field are assumed by the Assembler to be "dummy" symbols; the macro reference in the body of the source program must include an equivalent operand field. The Assembler will equate the macro reference's operand field to the MACRO directive's operand field and make substitutions accordingly.

This is how the substitution works:
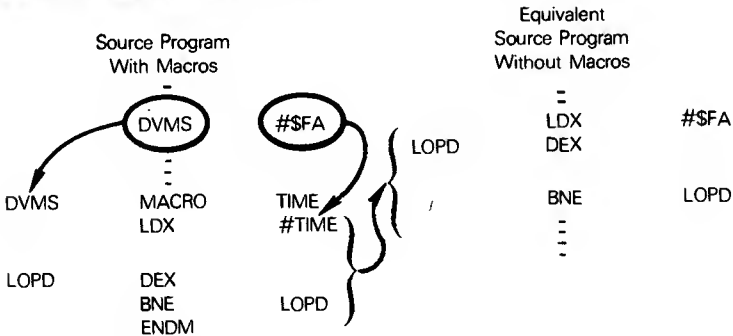
Depending on whose Assembler you are using, you can play interesting games with the macro parameter list: in theory (but not always in practice), there are no restrictions on the length or nature of the macro parameter list.

You will have to read the Assembler manual that accompanies your development system in order to know the exact macro features available to you.

INTERRUPTS

It would be hard to justify including interrupts within the microcomputer system developed in Chapter 4. In fact, interrupts should be used quite sparingly in microcomputer applications.

We will not enter into a long discussion on the strengths and weaknesses of interrupts within microcomputer systems — that subject has been adequately covered in "An Introduction To Microcomputers: Volume I — Basic Concepts". To summarize, however, recall that interrupts are a valid tool within microcomputer systems only when dealing with fast, asynchronous events.

Now having issued a warning against the indiscriminate use of interrupts, we will proceed to incorporate interrupt processing into our microcomputer program in the interests of demonstrating how it is done.

INTERRUPT HARDWARE CONSIDERATIONS

For an interrupt to be processed within an MC6800 microcomputer system, an interrupt request signal must be input high to the CPU at a time when interrupts have been enabled.

Interrupts are enabled and disabled by executing CLI and SEI instructions, respectively.

INTERRUPT ENABLE

Consider the simple case of a microcomputer system that includes one external interrupt. This is the "simple" case because it avoids vectoring and interrupt priority arbitration.

In this simple case, **when external logic requests an interrupt and the interrupt is acknowledged, the CPU saves the contents of the Program Counter, all registers and status on the stack;** then the address stored in memory locations FFF8$_{16}$ and FFF9$_{16}$ is loaded into the Program Counter. This may be illustrated as follows:

INTERRUPT ACKNOWLEDGE
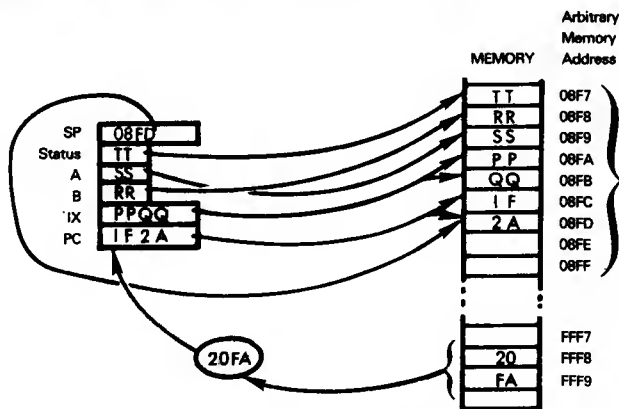
The illustration above uses arbitrary memory addresses. There is no specific significance to the memory addresses which have been selected, but they do make the illustration easier to follow. Given the memory addresses shown above, the interrupt is acknowledged following execution of an instruction whose object code was stored in one or more bytes of memory ending at location 1F29$_{16}$. The instruction which was about to be executed when the interrupt was acknowledged

has its object code stored in memory beginning at location $1F2A_{16}$. Following the interrupt acknowledge, the contents of all registers and status are stored at the top of the stack, as illustrated. Thus after all data has been pushed onto the stack, the new Stack Pointer contents will be $08F6_{16}$. The new address loaded into the Program Counter in the illustration above is $20FA_{16}$; thus program execution will resume, following the interrupt acknowledge, with an instruction sequence whose object code is stored in memory beginning at locations $20FA_{16}$.

The microcomputer system that we configured in Figure 4-2 uses the top 1,024 memory addresses to implement read-only memory. Thus, memory addresses $FFF8_{16}$ and $FFF9_{16}$ will select two bytes of read-only memory. Within these two bytes must be stored the beginning address for an instruction sequence which will be executed in response to any external interrupt — in our illustration $20FA_{16}$ is the required address. **This simple use of interrupt logic requires no special modifications to the microcomputer system as illustrated in Figure 4-2.** A simple connection to the interrupt request input of the MC6800 CPU is all that will be required. Other changes will occur strictly within the program that is stored in read-only memory.

Let us suppose that our microcomputer system performs a variety of background tasks in addition to enabling the print cycle, and the print cycle is initiated by an interrupt request. Interrupt logic replaces the PW READY signal. These are the changes which must be made to the program illustrated in Figure 4-6 in order to initiate the print cycle using an external interrupt:

```
              Old Program                              New Program
                  -                                        -
                  -                                        -
                  -                                        -
PRINT CYCLE PROGRAM                                        -
IN BETWEEN PRINT CYCLES TEST FFI                           -
(BIT 5 OF I/O PORT B) FOR A 0 VALUE                        -
START    LDA A    $C001                                    -
         AND A    #$20                                     -
         BNE      START                  ORG      START
INITIALIZE PRINT CYCLE. OUTPUT 0      INITIALIZE PRINT CYCLE. OUTPUT 0
TO BITS 0 AND 1 OF I/O PORT B.        TO BIT 0 OF I/O PORT B..
OUTPUT 1 TO BITS 2 AND 3 OF I/O       OUTPUT 1 TO BITS 2 AND 3 of I/O
PORT B                                PORT B
         LDA A    $C              START    LDA A    $C
         STA A    $C001                    STA A    $C001
OUTPUT 0 TO BIT 3 OF I/O PORT B.      OUTPUT 0 TO BIT 3 OF I/O PORT B.
THIS COMPLETES START RIBBON           THIS COMPLETES START RIBBON
MOTION                                MOTION
                  -                                        -
                  -                                        -
                  -                                        -
AT END OF PRINT CYCLE SET BIT 1 OF    AT END OF PRINT CYCLE RETURN
I/O PORT B TO 1 THIS SETS PW RDY      FROM INTERRUPT
HIGH                                      RTI
         LDA A    $C001            ORIGIN INTERRUPT SERVICE ROUTINE
         ORA A    #2                       ORG      $FFF8
         STA A    $C001                    FDB      START
         JMP      START                             -
                  -                                 -
                  -                                 -
                  -                                 -
                                                    -
```

The old program illustrated above is the program of Figure 4-6. Two changes must be made to this old program.

First of all the interrupt identifies the start of a new print cycle, so we can eliminate the "in between print cycles test".

Next we have eliminated the PW READY signal; therefore instructions which reset and subsequentially set this signal can be eliminated.

There is no significant change in the instruction sequence at the beginning of the print cycle since the same two instructions which reset PW READY remain to initially set and reset other signals. At the end of the old program, however, there were three instructions that set PW READY; these three instructions disappear in the new program.

Observe that the new program terminates with **a Return-from-Interrupt (RTI) instruction.** This instruction **restores the registers' and status contents which were saved when the interrupt was**

INTERRUPT RETURN

acknowledged; thus program logic branches back to the instruction which was about to get executed when the interrupt occurred — in our previous illustration the instruction stored at memory location $1F2A_{16}$.

Note that in the new program, the beginning address for the print cycle routine is stored in memory locations $FFF8_{16}$ and $FFF9_{16}$. This is done using the FDB directive following an ORG directive selecting address $FFF8_{16}$.

FDB ASSEMBLER DIRECTIVE

As you will see, program initialization instructions remain outside the interrupt service routine. Following the external interrupt, only the instructions which actually implement the print cycle get executed.

In all probability the initialization instructions which occur at the beginning of the program illustrated in Figure 4-6 will be executed following a Reset. The MC6800 handles a Reset as

RESET INTERRUPT

though it were a priority interrupt. The only difference is that the beginning address for the program which is to be executed following the Reset is fetched from memory locations $FFFE_{16}$ and $FFFF_{16}$, and contents of registers and status are not saved. If print cycle initialization instructions are executed following a Reset, this is how our program will look:

Old Program		
INDEX	EQU	$FF80
DELY	EQU	$FFEE
	ORG	0

SYSTEM RESET AND INITIALIZATION
FIRST OUTPUT CONTROL CODE TO I/O
PORT A CONTROL REGISTER

	CLR	$C002
		-
		-
		-

SET HAMMER PULSE, PW READY AND
PW REL HIGH. SET START RIBBON
MOTION LOW

	LDA A	#7
	STA A	$C001
		-
		-
		-

New Program		
INDEX	EQU	$FF80
DELY	EQU	$FFEE
	ORG	INIT

SYSTEM RESET AND INITIALIZATION
FIRST OUTPUT CONTROL CODE TO I/O
PORT A CONTROL REGISTER

INIT	CLR	$C002
		-
		-

SET HAMMER PULSE, PW REL HIGH
SET START RIBBON MOTION LOW

	LDA A	#5
	STA A	$C001
	JMP	SYSTEM
		-

Left column:

```
PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI
(BIT 5 OF I/O PORT B) FOR A 0
VALUE
START   LDA A   $C001
        AND A   #$20
        BNE     START
INITIALIZE PRINT CYCLE. OUTPUT 0
TO BITS 0 AND 1 OF I/O PORT B.
OUTPUT 1 TO BITS 2 AND 3 OF I/O
PORT B
        LDA A   $C
        STA A   $C001
OUTPUT 0 TO BIT 3 OF I/O PORT B.
THIS COMPLETES START RIBBON
MOTION
            -
            -
            -
AT END OF PRINT CYCLE SET BIT 1
OF I/O PORT B TO 1. THIS SETS
PW RDY HIGH
        LDA A   $C001
        ORA A   #2
        STA A   $C001
        JMP     START
            -
            -
            -
```

Right column:

```
INITIALIZE PRINT CYCLE. OUTPUT 0
TO BIT 0 OF I/O PORT B.
OUTPUT 1 TO BITS 2 AND 3 OF I/O
PORT B
START   LDA A   $C
        STA A   $C001
OUTPUT 0 TO BIT 3 OF I/O PORT B.
THIS COMPLETES START RIBBON
MOTION
            -
            -
            -
AT END OF PRINT CYCLE RETURN
FROM INTERRUPT
        RTI
ORIGIN INTERRUPT SERVICE ROUTINE
        ORG     $FFF8
        FDB     START
        ORG     $FFFE
        FDB     INIT
            -
```

So far as the print cycle interrupt service routine is concerned, nothing has changed. But the print cycle initialization instruction sequence has been converted into an independent interrupt service routine which gets executed following a Reset.

We have shown only the print cycle initialization instructions in the Reset interrupt service routine; in reality, a number of other initialization steps will be included, taking into account initialization procedures required by other logic supported by the microcomputer system.

MULTIPLE INTERRUPTS

What if your microcomputer system is connected to more than one external logic device that is capable of requesting interrupts? For example, a single MC6800 microcomputer system might be driving a number of printers. Without going into the economics of microcomputer multiple interrupt configurations, let us examine the ways in which multiple interrupts can be handled.

The one thing that changes when we go from single interrupts to multiple inter-rupts is the fact that the interrupt service routine is no longer unique. **There must be a different interrupt service routine for every external device capable of requesting an interrupt.** In turn that means that, following an interrupt acknowledge, we must have some means of knowing which interrupt service routine is to execute. Also, **if more than one device simultaneously requests interrupt service, which are we going to acknowledge — and in what order?** These are problems of interrupt vectoring and priority arbitration, subjects which have been covered in some detail in "An Introduction To Microcomputers: Volume I — Basic Concepts". We will not repeat discussion of these basic concepts in this book; rather we will look at practical ways in which multiple interrupts can be serviced within an MC6800 microcomputer system.

Conceptuelly the simplest method of handling multiple inter-rupts is to require every external device which is cepable of requesting an interrupt to elso have e buffer which cen be ec-cessed es e memory location. Within this buffer the external logic must store a status flag which indicates whether or not an interrupt has been requested. Now, following an interrupt acknowledge, we will first branch to a program that reads the contents of each external device buffer

in order of interrupt service priority, branching to the interrupt service routine for the first external device found to be requesting an interrupt. This is referred to as "polling". Here is a polling in-struction sequence:

```
        ORG     START
START OF INTERRUPT SERVICE ROUTINE
READ STATUS OF EACH EXTERNAL DEVICE CAPABLE OF
REQUESTING AN INTERRUPT
START   LDA A   DEV1
        BNE     PRG1
        LDA A   DEV2
        BNE     PRG2
        LDA A   DEV3
        BNE     PRG3
        -
        -
        -
        etc
```

The labels DEV1, DEV2, DEV3, etc., represent the memory addresses which select buffers of the individual external devices which are capable of requesting an interrupt. The labels PRG1, PRG2, PRG3, etc. identify the first instruction for the interrupt service routines corresponding to these ex-ternal devices. Device priorities are determined by the order in which devices are polled. Thus the device address DEV1 will have the highest priority.

The problem with polling is thet it tekes a fair amount of time to execute the poll-ing instruction sequence; end within the freme of reference of microprocessor device cost, the cost of heving buffers at the indlvidual devices — together with select logic for the buffers — may be quite significent.

But any MC6800 series support device that has interrupt logic also has buffers which mey be read in order to implement polling es a meens of handling multiple interrupt logic. For example, the MC6820 PIA, which we have already described, contains Control registers within which the two high order bits indicate whether interrupts have been re-quested via the CA1, CA2, CB1 and CB2 control lines. We have already seen how Control Register A is used to determine whether an interrupt request has occurred at an MC6820 PIA. If your microcomputer system is using such standard devices, then polling following an interrupt re-quest may have some merit. But if you are requesting interrupts via your own external logic, then polling has no merit. It will take no more logic to implement memory location $FFF8_{16}$ and/or $FFF9_{16}$ using an external buffer.

Figure 5-1 illustretes one scheme whereby memory loce-tion $FFF9_{16}$ selects en 8-to-3 line priority encoder and an 8-bit buffer. Devices cepeble of requesting interrupts elso in-put signels to the 8-to-3 encoder. Now following an interrupt acknowledge, the memory address which will be fetched from loca-

tions $FFF8_{16}$ and $FFF9_{16}$ varies with each possible external device that can request an interrupt.
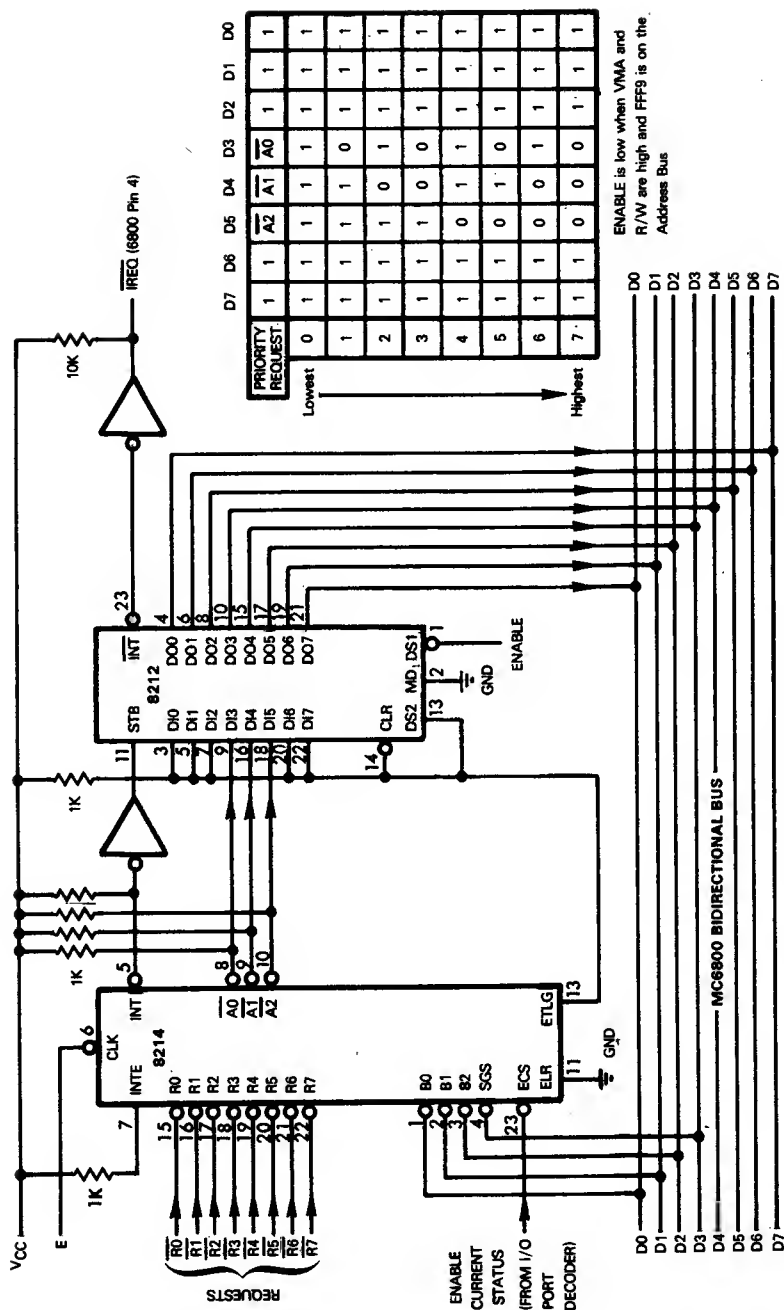
Figure 5-1. Interrupt Vectors Created Using 8-To-3 Encoder And 8-Bit Buffer

PRIORITY REQUEST	D7	D6	D5	D4	D3	D2	D1	D0
			$\overline{A2}$	$\overline{A1}$	$\overline{A0}$			
0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
2	1	1	1	1	0	1	1	1
3	1	1	1	1	0	1	1	1
4	1	1	1	0	1	1	1	1
5	1	1	1	0	1	1	1	1
6	1	1	0	0	1	1	1	1
7	1	1	0	0	0	1	1	1

Lowest → Highest

ENABLE is low when VMA and R/W are high and FFF9 is on the Address Bus

IREQ (6800 Pin 4)

10K

8212

STB   INT   DO0 DO1 DO2 DO3 DO4 DO5 DO6 DO7
DI0 DI1 DI2 DI3 DI4 DI5 DI6 DI7
DS2   MD, DS1   GND   ENABLE
CLR

MC6800 BIDIRECTIONAL BUS

D0 D1 D2 D3 D4 D5 D6 D7

1K

8214

CLK
INTE   INT
R0 R1 R2 R3 R4 R5 R6 R7
$\overline{A0}$ $\overline{A1}$ $\overline{A2}$
B0 B1 B2 SGS   ECS ELR   ETLG
GND

$\overline{R0}$ $\overline{R1}$ $\overline{R2}$ $\overline{R3}$ $\overline{R4}$ $\overline{R5}$ $\overline{R6}$ $\overline{R7}$

REQUESTS

ENABLE CURRENT STATUS (FROM I/O PORT DECODER)

VCC
E

The most interesting aspect of Figure 5-1 is the fact that it is exactly the same logic that was used to generate a Restart instruction for the 8080A microcomputer system described in Chapter 5 of "8080 Programming For Logic Design". The 8-to-3 priority encoder is an 8214 device; the 8212 is an 8-bit buffer. Both of these devices have been described in detail in Chapter 4 of "An Introduction To Microcomputers: Volume II — Some Real Products".

Eight external devices are capable of requesting interrupts. These eight devices input individual signals to the $\overline{R0}/\overline{R7}$ inputs of the 8214 device. The inputs are wire-ORed to provide a master interrupt request. The 8214 device is enabled by a clock signal that makes a low-to-high transition. This clock signal is created by the E synchronization signal of the MC6800 microcomputer system. The three outputs of the 8214 decoder are input to an 8212 8-bit buffer, the remaining inputs of which are tied to +5V. This I/O port is selected by the memory address $FFF9_{16}$. Thus, eight different memory addresses will be read for the eight possible external interrupts as follows:


```
MEMORY
LOCATION          X X
FFF8₁₆
```
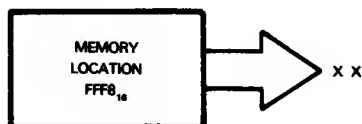
(X represents any hexadecimal digit)


```
                  C7
                  CF
                  D7
LOGIC OF          DF
FIGURE 5-1        E7
                  EF
                  F7
                  FF
```
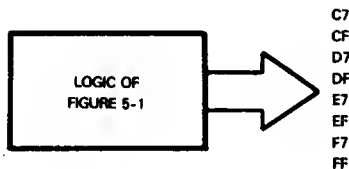
Device	Service Routine Origin
R0	X X F F
R1	X X F 7
R2	X X E F
R3	X X E 7
R4	X X D F
R5	X X D 7
R6	X X C F
R7	X X C 7

Figure 5-2 shows an alternative configuration which uses an MC6828 Priority Interrupt Controller (PIC) in order to provide vectored priority Interrupts. The MC6828 Priority Interrupt Controller has been described in detail in "An Introduction To Microcomputers: Volume II — Some Real Products", Chapter 6.

<table>
<tr><td>MC6828
PRIORITY
INTERRUPT
CONTROLLER</td></tr>
</table>
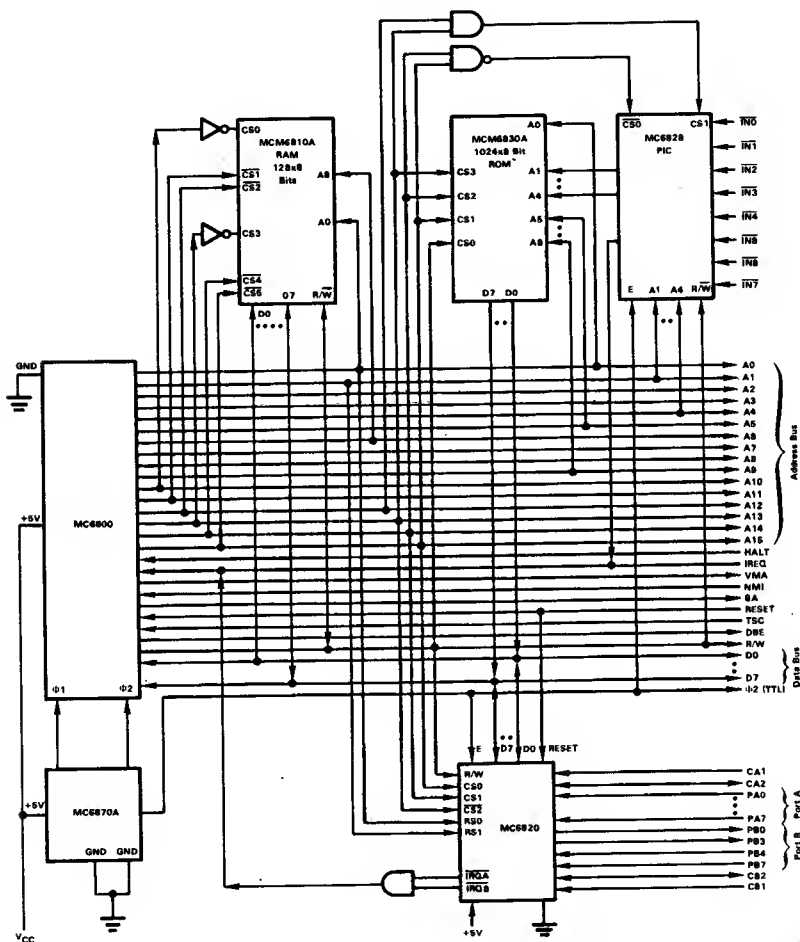
Figure 5-2. Interrupt Vectors Created Using An MC6828 Priority Interrupt Controller

The MC6828 PIC is positioned serially, preceding the external memory device which is to be selected by the addresses $FFF8_{16}$ and $FFF9_{16}$. Address lines A1, A2, A3 and A4 terminate at the MC6828. Logic within the MC6828 appropriately manipulates these four address lines and outputs some value which may differ from the input value. This may be illustrated as follows:

Address
transmitted
by CPU

Address
received
by memory

A15	1		1
A14	1		1
A13	1		1
A12	1		1
A11	1		1
A10	1		1
A9	1		1
A8	1		1
A7	1		1
A6	1		1
A5	1		1
A4	1		Y
A3	1		Y
A2	0		Y
A1	0		Y
A0	X		X

Address lines modified by MC6828
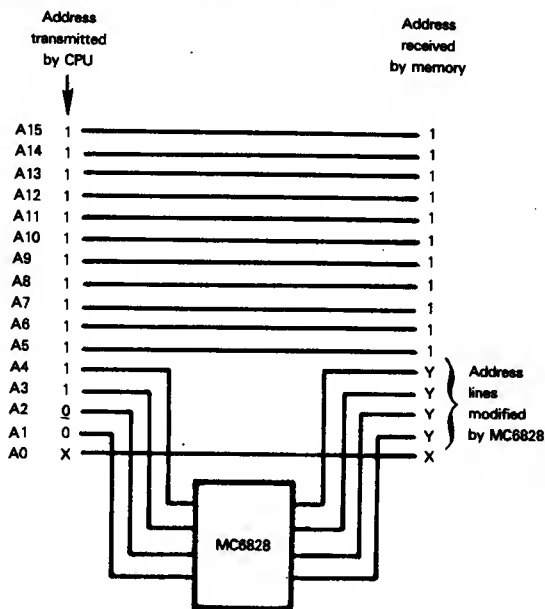
MC6828

Thus, what the MC6828 does is extend the two addresses $FFF8_{16}$ and $FFF9_{16}$ into 16 addresses, $FFE8_{16}$ through $FFF7_{16}$.

Conceptually, the MC6828 is acting as an 8-way switch. The CPU addresses the switch by its "stem", via a single address. The actual conduit for the transfer of two bytes of data depends on the switch position at the time the CPU accesses the switch stem; and the switch position is going to be determined by the highest priority active interrupt request. This may be illustrated as follows:

IN7	FFF6, FFF7
IN6	FFF4, FFF5
IN5	FFF2, FFF3
IN4	FFF0, FFF1
IN3	FFEE, FFEF
IN2	FFEC, FFED
IN1	FFEA, FFEB
IN0	FFE8, FFE9

FFF8, FFF9
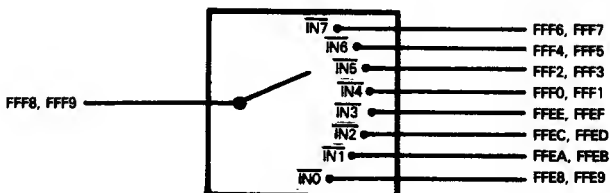
Comparing Figures 5-2 and 5-1 we see that there is a fundamental philosophical difference between the ways in which interrupt vectors have been created. The MC6828 device in Figure 5-2 modifies the address which ultimately arrives at memory devices selected by $FFF8_{16}$ and $FFF9_{16}$. Eight addresses must be maintained in 16 bytes

of external read/write memory: all 16 bytes of external read/write memory are selected by the same two addresses output from the CPU; logic of the MC6828 PIC determining which two of the 16 bytes will in fact be selected.

In contrast, logic of Figure 5-1 is selected by a single memory address, but the data which is read from this single memory address is varied.

Table 5-2. MC6828 Address Vectors Created For Eight
Priority Interrupt Requests

PRIORITY	PIN	Z4	Z3	Z2	Z1	EFFECTIVE ADDRESSES
Highest 7	$\overline{IN7}$	1	0	1	1	FFF6 and FFF7
6	$\overline{IN6}$	1	0	1	0	FFF4 and FFF5
5	$\overline{IN5}$	1	0	0	1	FFF2 and FFF3
4	$\overline{IN4}$	1	0	0	0	FFF0 and FFF1
3	$\overline{IN3}$	0	1	1	1	FFEE and FFEF
2	$\overline{IN2}$	0	1	1	0	FFEC and FFED
1	$\overline{IN1}$	0	1	0	1	FFEA and FFEB
Lowest 0	$\overline{IN0}$	0	1	0	0	FFE8 and FFE9

Table 5-2 defines the priorities that will be applied to simultaneous interrupt requests occurring at pins $\overline{IN0}$ - $\overline{IN7}$. This table also indicates the exact memory addresses which

will be created by the MC6828 in response to each of the interrupt requests. In order to use the MC6828 PIC on an MC6800 microcomputer system, 16 bytes of PROM or ROM, selected by the addresses given in Table 5-2, must be connected to the MC6828. Within these 16 bytes of PROM or ROM, you must store the starting addresses for the eight interrupt service routines which are going to be executed following acknowledgement of each possible external interrupt request. For example, suppose that interrupt requests arriving at the $\overline{IN5}$ pin of the MC6828 must be serviced by an interrupt service routine whose first executable instruction is stored in memory location $2E00_{16}$. The value $2E00_{16}$ must then be stored in the two PROM or ROM bytes selected by memory addresses $FFF2_{16}$ and $FFF3_{16}$. Remember, the high order byte of an address is always stored at the lower address. Thus $2E_{16}$ will be stored in memory location $FFF2_{16}$ while 00_{16} is stored in memory location $FFF3_{16}$.

The MC6828 provides a very elementary level of interrupt inhibit logic. You can output a mask to the MC6828 identifying a priority level below which all interrupts will be inhibited.

Now the mask is written out to the MC6828 in a very unusual way.

Recall that the MC6828 requires memory addresses $FFE8_{16}$ through $FFF9_{16}$ to access PROM or ROM. Any attempt to write into these memory addresses will be ignored. The MC6828 takes advantage of this fact by trapping attempts to write into memory locations $FFE8_{16}$ through $FFF9_{16}$.

That is to say, when R/W is low while $\overline{CS0}$ is low and CS1 is high, the MC6828 considers itself selected, but it interprets the four address lines A1, A2, A3, A4 as data, defining the mask level below which interrupts will be inhibited. **Table 5-3 defines the way in which the mask specified by address lines A1, A2, A3 and A4 will be interpreted.**

Table 5-3. MC6828 Interrupt Masks — Their Creation And Interpretation

Write anything to this address:	and Address Bus lines A1-A4 will have this value:	Which will inhibit all interrupts, including and below:
FFE0 or FFE1	0000	All interrupts enabled
FFE2 or FFE3	0001	IN1
FFE4 or FFE5	0010	IN2
FFE6 or FFE7	0011	IN3
FFE8 or FFE9	0100	IN4
FFEA or FFEB	0101	IN5
FFEC or FFED	0110	IN6
FFEE or FFEF	0111	IN7
FFE0 through FFFF	1000 through 1111	All interrupts disabled

JUSTIFYING INTERRUPTS

Minicomputer programmers and large computer programmers make indiscriminate use of interrupts simply to share the cost of the Central Processing Unit among a number of different applications.

You, as a microcomputer user, are going to have to justify sharing a cost | **INTERRUPT ECONOMICS** which may range between $5 and $20. Against this cost you must charge the cost of external logic needed to create interrupt request signals — as well as the extra cost of programming. **The economic tradeoff makes it far from obvious that interrupts are viable within microcomputer systems.** You must examine your application with care before assuming out of hand that interrupts represent the way to go. A second CPU, or an entire second microcomputer system will frequently be cheaper than using interrupts to share a single microcomputer system between a number of different applications.

Assuming that interrupts look economical for your application, timing considerations are also important. | **INTERRUPT TIMING CONSIDERATIONS**

Certainly interrupts look very attractive when your application is handling asynchronous events. In our case, **suppose the average print cycle lasts approximately 10 milliseconds; also, suppose it is impossible to say whether the time interval between print cycles will be 1 millisecond or 100 milliseconds. Under these circumstances, in order to execute some other program in the time in between print cycles, we must use interrupts to initiate the print cycle** — since we have no idea when the next print cycle is to begin.

In reality, the time which elapses between print cycles will be very accurately known. **A printer will have some advertised character printing rate.** If this rate is 45 characters per second, then 22.2 milliseconds will be required per printed character. If 10 of the 22 milliseconds are needed to execute the actual print cycle routine, then 12 milliseconds will remain in between print cycles. **We no longer need interrupts.** So long as the program which executes in between print cycles is broken into segments, each of which executes in 12 milliseconds or less, then each segment can terminate with an instruction loop which tests the status of the velocity decode input in order to initiate the next print cycle:

```
START   LDA A   $C001      INPUT I/O PORT B TO ACCUMULATOR A
        AND A   #$20       ISOLATE BIT 5
        BNE     START      IF NOT 0, RETURN TO START
```

Chapter 6
THE MC6800 INSTRUCTION SET

Instructions falsely frighten microcomputer users who are new to programming. Taken as an isolated event, operations associated with the execution of a single instruction are easy enough to follow — and that is the purpose of this chapter.

Why are the instructions of a microcomputer referred to as an instruction "set"? Because the instructions selected by the designers of any microcomputer are selected with great care; it must be easy to execute complex operations as a sequence of simple events — each of which is represented by one instruction from a well designed instruction "set".

Remaining consistent with "An Introduction To Microcomputers: Volume II", Table 6-1 summarizes the MC6800 microcomputer instruction set, with similar instructions grouped together.

Individual instructions are described next in alphabetic order of instruction mnemonic.

In addition to simply stating what each instruction does, the purpose of the instruction within normal programming logic is identified.

ABBREVIATIONS

These are the abbreviations used in this chapter:

ACX   Either Accumulator A or Accumulator B

The registers:

A,B	Accumulator
X	Index register
PC	Program Counter
SP	Stack Pointer
SR	Status register

Statuses shown:

C	Carry status
Z	Zero status
S	Sign status
O	Overflow status
I	Interrupt status
A_C	Auxiliary Carry status

Symbols in the STATUSES column:

(blank)	operation does not affect status
X	operation affects status
0	flag is cleared by the operation
1	flag is set by the operation

ADR8   An 8-bit (1-byte) quantity which may be used to directly address the first 256 locations in memory, or may be an 8-bit unsigned displacement to be added to the Index register.

ADR16   A 16-bit memory address

B2	Instruction Byte 2
B3	Instruction Byte 3
DATA	An 8-bit binary data unit
DATA16	A 16-bit binary data unit
DISP	An 8-bit signed binary address displacement
xx(HI)	The high order 8 bits of the 16-bit quantity xx; for example, SP(HI) means bits 15 - 8 of the Stack Pointer.
xx(LO)	The low order 8 bits of the 16-bit quantity xx; for example, PC(LO) means bits 7 - 0 of the Program Counter.
[ ]	Contents of location enclosed with brackets.
[[ ]]	Implied memory addressing; the contents of the memory location designated by the contents of a register.
[MEM]	Symbol for memory location indicated by base page direct, extended direct; or indexed addressing.

That is:

$$[MEM] = [ADR8]$$

or

$$[ADR16]$$

or

$$[[X] + ADR8]$$

[M]	Symbol for memory location indicated by extended direct or indexed addressing. That is:

$$[M] = [ADR16]$$

or

$$[[X] + ADR8]$$

Λ	Logical AND
V	Logical OR
⩛	Logical Exclusive-OR
←	Data is transferred in the direction of the arrow.

CONDITION CODES

The six condition codes are stored in a Condition Code register as follows:
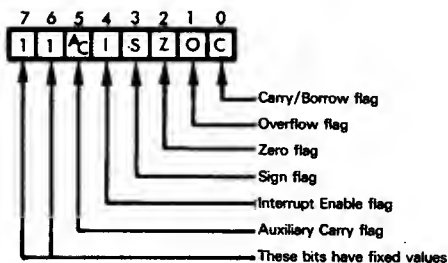
The effect of instruction execution on status is illustrated in the following way:
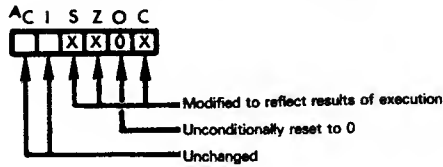
Within instruction execution illustrations, an X identifies a status that is set or reset. A 0 identifies a status that is always cleared. A blank means the status does not change.

STATUS CHANGES WITH INSTRUCTION EXECUTION

INSTRUCTION OBJECT CODES

Instruction object codes are represented as 2 hexadecimal digits for instructions without variations.

Instruction object codes are represented as 8 binary digits for instructions with variations; the binary digit representation of variations is then identifiable.

INSTRUCTION EXECUTION TIMES AND CODES

Table 6-2 lists instructions in alphabetic order, showing object codes and execution times, expressed as machine cycles.

Table 6-1. A Summary Of The MC6800 Instruction Set

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES						OPERATION PERFORMED
				C	Z	S	O	Ac	I	
PRIMARY MEMORY REFERENCE AND I/O	LDA	ACX,ADR6 ACX,ADR18	2 3		X	X	0			[ACX]←[MEM] Load A or B using base page direct, extended direct, or indexed addressing.
	STA	ACX,ADR8 ACX,ADR16	2 3		X	X	0			[MEM]←[ACX] Store A or B using direct, extended, or indexed addressing.
	LDX	ADR8 ADR16	2 3		X	X	0			[X(HI)]←[MEM], [X(LO)]←[MEM + 1] Load Index register using direct, extended, or indexed addressing. Sign status reflects Index register bit 15.
	STX	ADR6 ADR16	2 3		X	X	0			[MEM]←[X(HI)], [MEM + 1]←[X(LO)] Store contents of Index register using direct, extended, or indexed addressing. Sign status reflects Index register bit 15.
	LDS	ADR6 ADR18	2 3		X	X	0			[SP(HI)]←[MEM], [SP(LO)]←[MEM + 1] Load Stack Pointer using direct, extended, or indexed addressing. Sign status reflects Stack Pointer bit 15.
	STS	ADR6 ADR18	2 3		X	X	0			[MEM]←[SP(HI)], [MEM + 1]←[SP(LO)] Store contents of Stack Pointer using direct, extended, or indexed addressing. Sign status reflects Stack Pointer bit 15.
SECONDARY MEMORY REFERENCE (MEMORY OPERATE)	ADD	ACX,ADR8 ACX,ADR16	2 3	X	X	X	X	X		[ACX]←[ACX] + [MEM] Add to Accumulator A or B using base page direct, extended direct, or indexed addressing.
	ADC	ACX,ADR8 ACX,ADR16	2 3	X	X	X	X	X		[ACX]←[ACX] + [MEM] + C Add with carry to Accumulator A or B using direct, extended, or indexed addressing.
	AND	ACX,ADR8 ACX,ADR16	2 3		X	X	0			[ACX]←[ACX] ∧ [MEM] AND with Accumulator A or B using direct, extended, or indexed addressing.
	BIT	ACX,ADR6 ACX,ADR16	2 3		X	X	0			[ACX] ∧ [MEM] AND with Accumulator A or B, but only Status register is effected.
	CMP	ACX,ADR6 ACX,ADR16	2 3	X	X	X	X			[ACX] - [MEM] Compare with Accumulator A or B (only Status register is affected).
	EDR	ACX,ADR8 ACX,ADR16	2 3		X	X	0			[ACX]←[ACX]⊕[MEM] Exclusive-OR with Accumulator A or B using direct, extended, or indexed addressing.
	ORA	ACX,ADR6 ACX,ADR16	2 3		X	X	0			[ACX]←[ACX] ∨ [MEM] OR with Accumulator A or B using direct, extended, or indexed addressing.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES						OPERATION PERFORMED
				C	Z	S	O	Ac	I	
SECONDARY MEMORY REFERENCE [MEMORY OPERATE] CONTINUED	SUB	ACX.ADR8	2	×	×	×	×			[ACX]←[ACX] - [MEM] Subtract from Accumulator A or B using direct, extended, or indexed addressing.
		ACX.ADR16	3							
	SBC	ACX.ADR8	2	×	×	×	×			[ACX]←[ACX] - [MEM] - C Subtract with carry from Accumulator A or B using direct, extended, or indexed addressing.
		ACX.ADR16	3							
	CPX	ADR8	2		×	×				[X(HI)] - [MEM], [X(LO)] - [MEM + 1] Compare with contents of Index register (only Status register is affected). Sign and Overflow statuses reflect result on most significant byte.
		ADR16	3							
	CLR	ADR8	2	0	1	0	0			[M]←00₁₆ Clear memory location using extended or indexed addressing.
		ADR16	3							
	COM	ADR8	2	1	×	×	0			[M]←[M] Complement contents of memory location (ones complement).
		ADR18	3							
	NEG	ADR8	2	×	×	×	×			[M]←00₁₆ - [M] Negate contents of memory location (twos complement). Carry status is set if result is 00₁₆, and reset otherwise. Overflow status is set if result is 80₁₆ and reset otherwise.
		ADR18	3							
	DEC	ADR8	2		×	×	×			[M]←[M] - 1 Decrement contents of memory location, using extended or indexed addressing. Overflow status is set if operand was 80₁₆ before execution, and cleared otherwise.
		ADR16	3							
	INC	ADR8	2		×	×	×			[M]←[M] + 1 Increment contents of memory location, using extended or indexed addressing. Overflow status is set if operand was 7F₁₆ before execution, and cleared otherwise.
		ADR16	3							
	ROL	ADR8	2	×	×	×	×			[C]←[7 ────── 0]←, 0←S·C [M] Rotate contents of memory location left through carry.
		ADR16	3							
	ROR	ADR8	2	×	×	×	×			[C]→[7 ────── 0]→, 0←S·C [M] Rotate contents of memory location right through carry.
		ADR16	3							

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	C	Z	S	O	Ac	I	OPERATION PERFORMED
SECONDARY MEMORY REFERENCE (MEMORY OPERATE) CONTINUED	ASL	ADR8 ADR16	2 3	×	×	×	×			C ← [7...0] ← 0, D→S⊕C Arithmetic shift left. Bit 0 is set to 0.
	ASR	ADR8 ADR16	2 3	×	×	×	×			[M] → C, 0→S⊕C Arithmetic shift right. Bit 7 stays the same.
	LSR	ADR8 ADR16	2 3	×	×	0	×			0 → [M] → C, 0→S⊕C Logical shift right. Bit 7 is set to 0.
	TST	ADR8 ADR16	2 3	0	×	×	0			[M] - 00₁₆. Test contents of memory location for zero or negative value.
IMMEDIATE	LDA	ACX,DATA	2		×	×	0			[ACX]←DATA. Load A or B immediate.
	LDX	DATA16	3		×	×	0			[X(HI)]←[B2], [X(LO)]←[B3]. Load Index register immediate. Sign status reflects Index register bit 15.
	LDS	DATA16	3		×	×	0			[SP(HI)]←[B2], [SP(LO)]←[B3]. Load Stack Pointer immediate. Sign status reflects Stack Pointer bit 15.
IMMEDIATE OPERATE	ADD	ACX,DATA	2	×	×	×	×	×		[ACX]←[ACX]+DATA. Add immediate to Accumulator A or B.
	ADC	ACX,DATA	2	×	×	×	×	×		[ACX]←[ACX]+DATA+C. Add immediate with carry to Accumulator A or B.
	AND	ACX,DATA	2		×	×	0			[ACX]←[ACX]∧DATA. AND immediate with Accumulator A or B.
	BIT	ACX,DATA	2		×	×	0			[ACX]∧DATA. AND immediate with Accumulator A or B, but only the Status register is effected.

STATUSES: C Z S O Ac I

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	C	Z	S	O	Ac	I	OPERATION PERFORMED
IMMEDIATE OPERATE (CONTINUED)	CMP	ACX,DATA	2	X	X	X	X			[ACX] - DATA Compare immediate with Accumulator A or B (only the Status register is affected).
	EOR	ACX,DATA	2		X	X	0			[ACX]←[ACX]⊕DATA Exclusive-OR immediate with Accumulator A or B.
	DRA	ACX,DATA	2		X	X	0			[ACX]←[ACX] V DATA OR immediate with Accumulator A or B.
	SUB	ACX,DATA	2	X	X	X	X			[ACX]←[ACX] - DATA Subtract immediate from Accumulator A or B.
	SBC	ACX,DATA	2	X	X	X	X			[ACX]←[ACX] - DATA - C Subtract immediate with carry from Accumulator A or B.
	CPX	DATA16	3		X	X	X			[X(HI)] - [B2], [X(LO)] - [B3] Compare immediate with contents of index register (only the Status register is effected). Sign and Overflow status reflect result on most significant byte.
JUMP	JMP	ADR8 ADR16	2 3							[PC]←[X] + ADR8 or [PC(HI)]←[B2], [PC(LO)]←[B3] Jump to indexed or extended address.
	JSR	ADR6 ADR16	2 3							[[SP]]←[PC(LO)], [[SP]-1]←[PC(HI)], [SP]←[SP]-2 [PC]←[X] + ADR8 or [PC(HI)]←[B2], [PC(LO)]←[B3] Jump to subroutine (indexed or extended addressing).
	BRA	DISP	2							[PC]←[PC] + DISP + 2 Unconditional branch relative to present Program Counter contents.
	BSR	DISP	2							[[SP]]←[PC(LO)], [[SP]-1]←[PC(HI)], [SP]←[SP]-2, [PC]←[PC] + DISP + 2 Unconditional branch to subroutine located relative to present Program Counter contents.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	C	Z	S	O	Ac	I	OPERATION PERFORMED
BRANCH ON CONDITION	BCC	DISP	2							[PC]→[PC]+DISP+2 if the given condition is true:
	BCS	DISP	2							$\quad$ C = 0 (Branch if carry clear)
	BEQ	DISP	2							$\quad$ C = 1 (Branch if carry set)
	BGE	DISP	2							$\quad$ Z = 1 (Branch if equal to zero)
	BGT	DISP	2							$\quad$ S⊕O = 0 (Branch if greater than or equal to zero)
	BHI	DISP	2							$\quad$ Z∨(S⊕O) = 0 (Branch if greater than zero)
	BLE	DISP	2							$\quad$ C∨Z = 0 (Branch if Accumulator contents higher than comparand)
	BLS	DISP	2							$\quad$ Z∨(S⊕O) = 1 (Branch if less than or equal to zero)
	BLT	DISP	2							$\quad$ C∨Z = 1 (Branch if Accumulator contents less than or same as comparand)
	BMI	DISP	2							$\quad$ S⊕O = 1 (Branch if less than zero)
	BNE	DISP	2							$\quad$ S = 1 (Branch if minus)
	BVC	DISP	2							$\quad$ Z = 0 (Branch if not equal to zero)
	BVS	DISP	2							$\quad$ O = 0 (Branch if overflow clear)
	BPL	DISP	2							$\quad$ O = 1 (Branch if overflow set)
										$\quad$ S = 0 (Branch if plus)
REGISTER-REGISTER MOVE	TAB		1		X	X	0			[B]←[A] Move Accumulator A contents to Accumulator B.
	TBA		1		X	X	0			[A]←[B] Move Accumulator B contents to Accumulator A.
	TXS		1							[SP]←[X]-1 Move Index register contents to Stack Pointer and decrement.
	TSX		1							[X]←[SP]+1 Move Stack Pointer contents to Index register and increment.
REGISTER OPERATE	ABA		1	X	X	X	X	X		[A]←[A]+[B] Add contents of Accumulators A and B.
	CBA		1	X	X	X	X			[A]-[B] Compare contents of Accumulators A and B. Only the Status register is affected.
	SBA		1	X	X	X	X			[A]←[A]-[B] Subtract contents of Accumulator B from those of Accumulator A.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	C	Z	S	O	A_C	I	OPERATION PERFORMED
				STATUSES						
REGISTER OPERATE	CLR	ACX	1	0	1	0	0			[ACX] → 00₁₆ Clear Accumulator A or B.
	CDM	ACX	1	1	x	x	0			[ACX] → $\overline{[ACX]}$ Complement contents of Accumulator A or B (ones complement).
	NEG	ACX	1	x	x	x	x			[ACX] → 00₁₆ - [ACX] Negate contents of Accumulator A or B (twos complement). Carry status is set if result is 00₁₆ and reset otherwise. Overflow status is set if result is 80₁₆ and reset otherwise.
	DAA	ACX	1	x	x	x	x			Decimal adjust A. Convert contents of A (the binary sum of BCD operands) to BCD format. Carry status is set if value of upper four bits is greater than 9, but not cleared if previously set.
	DEC	ACX	1		x	x	x			[ACX] → [ACX] - 1 Decrement contents of Accumulator A or B. Overflow status is set if operand was 80₁₆ before execution, and cleared otherwise.
	DEX		1		x					[X] → [X] - 1 Decrement contents of Index register.
	DES		1							[SP] → [SP] - 1 Decrement contents of Stack Pointer.
	INC	ACX	1		x	x	x			[ACX] → [ACX] + 1 Increment contents of Accumulator A or B. Overflow status is set if operand was 7F₁₆ before execution, and cleared otherwise.
	INX		1		x					[X] → [X] + 1 Increment contents of Index register.
	INS		1							[SP] → [SP] + 1 Increment contents of Stack Pointer.
	ROL	ACX	1	x	x	x	x			[ACX] Rotate Accumulator A or B left through carry.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	C	Z	S	O	Ac	I	OPERATION PERFORMED
REGISTER OPERATE (CONTINUED)	ROR	ACX	1	X	X	X	X			[ACX] — Rotate Accumulator A or B right through carry. $0 \rightarrow S \rightarrow C$
	ASL	ACX	1	X	X	X	X			[ACX] — Arithmetic shift left. Bit 0 is set to 0. $0 \rightarrow S \rightarrow C$
	ASR	ACX	1	X	X	X	X			[ACX] — Arithmetic shift right. Bit 7 stays the same. $0 \rightarrow S \rightarrow C$
	LSR	ACX	1	X	X	0	X			[ACX] — Logical shift right. Bit 7 is set to 0. $0 \rightarrow S \rightarrow C$
	TST	ACX	1	0	X	X	0			$[ACX] - 00_{16}$ — Test contents of Accumulator A or B for zero or negative value.
STACK	PSH	ACX	1							$[[SP]] \rightarrow [ACX]$ $[SP] \rightarrow [SP] - 1$ Push contents of Accumulator A or B onto top of Stack and decrement Stack Pointer.
	PUL	ACX	1							$[SP] \rightarrow [SP] + 1$ $[ACX] \rightarrow [[SP]]$ Increment Stack Pointer and pull Accumulator A or B from top of Stack.
	RTS		1							$[PC(HI)] \rightarrow [[SP] + 1], [PC(LO)] \rightarrow [[SP] + 2], [SP] \rightarrow [SP] + 2$ Return from subroutine. Pull PC from top of Stack and increment Stack Pointer.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	Z	S	O	A_C	I	OPERATION PERFORMED
INTERRUPT	CLI		1						0	$I \rightarrow 0$ Clear interrupt mask to enable interrupts.
	SEI		1						1	$I \rightarrow 1$ Set interrupt mask to disable interrupts.
	RTI		1	X	X	X	X	X	X	$[SR] \rightarrow [[SP]+1]$, $[B] \rightarrow [[SP]+2]$, $[A] \rightarrow [[SP]+3]$, $[X(HI)] \rightarrow [[SP]+4]$, $[X(LO)] \rightarrow [[SP]+5]$, $[PC(HI)] \rightarrow [[SP]+6]$, $[PC(LO)] \rightarrow [[SP]+7]$, $[SP] \rightarrow [SP]+7$ Return from interrupt. Pull registers from Stack and increment Stack Pointer.
	SWI		1						1	$[[SP]] \rightarrow [PC(LO)]$, $[[SP]-1] \rightarrow [PC(HI)]$, $[[SP]-2] \rightarrow [X(LO)]$, $[[SP]-3] \rightarrow [X(HI)]$, $[[SP]-4] \rightarrow [A]$, $[[SP]-5] \rightarrow [B]$, $[[SP]-6] \rightarrow [SR]$, $[SP] \rightarrow [SP]-7$, $[PC(HI)] \rightarrow [FFFA_{16}]$ $[PC(LO)] \rightarrow [FFFB_{16}]$ Software Interrupt: push registers onto Stack, decrement Stack Pointer, and jump to interrupt subroutine.

Table 6-1. A Summary Of The MC6800 Instruction Set (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES						OPERATION PERFORMED
				C	Z	S	O	AC	I	
INTERRUPT (CONTINUED)	WAI		1						1	[[SP]]—[PC(LO)], [[SP]-1]—[PC(HI)], [[SP]-2]—[X(LO)], [[SP]-3]—[X(HI)], [[SP]-4]—[A], [[SP]-5]—[B], [[SP]-6]—[SR], [SP]—[SP]-7 Push registers onto Stack, decrement Stack Pointer, and wait for interrupt. If [I]=1 when WAI is executed, a non-maskable interrupt is required to exit the Wait state. Otherwise, [I]—1 when the interrupt occurs.
STATUS	CLC		1	0						C —0 . Clear carry
	SEC		1	1						C —1 Set carry
	CLV		1				0			O —0 Clear overflow status bit
	SEV		1				1			O —1 Set overflow status bit
	TAP		1	X	X	X	X	X	X	[SR]—[A] Transfer contents of Accumulator A to Status register.
	TPA		1							[A]—[SR] Transfer contents of Status register to Accumulator A.
	NOP		1							No Operation

The following codes are used in Table 6-2:

aa   two bits choosing the address mode:

    00   immediate data
    01   base page direct addressing
    10   indexed addressing
    11   extended direct addressing

pp   the second byte of a two- or three-byte instruction.

qq   the third byte of a three-byte instruction.

x   one bit choosing the Accumulator:

    0   Accumulator A
    1   Accumulator B

yy   two bits choosing the address mode:

    00   (inherent addressing) Accumulator A
    01   (inherent addressing) Accumulator B
    10   indexed addressing
    11   extended direct addressing

y   one bit choosing the address mode:

    0   indexed addressing
    1   extended direct addressing

Two numbers in the "Machine Cycles" column (for example, 2 - 5) indicate that execution time depends on the addressing mode.

Table 6-2. MC6800 Instruction Set Object Codes

MNEMONIC	OPERAND(S)	OBJECT CODE	BYTE	MACHINE CYCLES
ABA		1B	1	2
ADC	ACX,	1xaa1001		
	ADR6 or DATA	pp	2	2-5
	ADR16	qq	3	4
ADD	ACX,	1xaa1011		
	ADR8 or DATA	pp	2	2-5
	ADR18	qq	3	4
AND	ACX,	1xaa0100		
	ADR6 or DATA	pp	2	2-5
	ADR16	qq	3	4
ASL	ACX	01yy1000	1	2
	ADR8	pp	2	7
	ADR16	qq	3	6
ASR	ACX	01yy0111	1	2
	ADR8	pp	2	7
	ADR16	qq	3	6
BCC	DISP	24 pp	2	4
BCS	DISP	25 pp	2	4
BEQ	DISP	27 pp	2	4
BGE	DISP	2C pp	2	4
BGT	DISP	2E pp	2	4
BHI	DISP	22 pp	2	4
BIT.	ACX,	1xaa0101		
	ADR8 or DATA	pp	2	2-5
	ADR16	qq	3	4
BLE	DISP	2F pp	2	4
BLS	DISP	23 pp	2	4
BLT	DISP	2D pp	2	4
BMI	DISP	2B pp	2	4
BNE	DISP	26 pp	2	4
BPL	DISP	2A pp	2	4
BRA	DISP	20 pp	2	4
BSR	DISP	8D pp	2	8
BVC	DISP	28 pp	2	4
BVS	DISP	29 pp	2	4
CBA		11	1	2
CLC		0C	1	2
CLI		0E	1	2
CLR	ACX	01yy1111	1	2
	ADR8	pp	2	7
	ADR16	qq	3	8
CLV		0A	1	2
CMP	ACX,	1xaa0001		
	ADR8 or DATA	pp	2	2-5
	ADR18	qq	3	4
COM	ACX	01yy0011	1	2
	ADR8	pp	2	7
	ADR16	qq	3	6
CPX		10aa1100		
	ADR6	pp	2	4-6
	ADR18 or DATA16	qq	3	3-5
DAA		19	1	2
DEC	ACX	01yy1010	1	2
	ADR8	pp	2	7
	ADR18	qq	3	6

Table 6-2. MC6800 Instruction Set Object Codes (Continued)

MNEMONIC	OPERAND(S)	OBJECT CODE	BYTE	MACHINE CYCLES
DES		34	1	4
DEX		09	1	4
EOR	ACX,	1xaa1000		
	ADR8 or DATA	pp	2	2-5
	ADR16	qq	3	4
INC	ACX	01yy1100	1	2
	ADR8	pp	2	7
	ADR16	qq	3	8
INS		31	1	4
INX		08	1	4
JMP		011y1110		
	ADR8	pp	2	4
	ADR16	qq	3	3
JSR		101y1101		
	ADR8	pp	2	8
	ADR16	qq	3	9
LDA	ACX,	1xaa0110		
	ADR8 or DATA	pp	2	2-5
	ADR18	qq	3	4
LDS		10aa1110		
	ADR8	pp	2	3-5
	ADR16 or DATA16	qq	3	4-6
LDX		11aa1110		
	ADR8	pp	2	3-5
	ADR18 or DATA18	qq	3	4-6
LSR	ACX	01yy0100	1	2
	ADR8	pp	2	7
	ADR18	qq	3	8
NEG	ACX	01yy0000	1	2
	ADR8	pp	2	7
	ADR18	qq	3	6
NOP		01	1	2
ORA	ACX,	1xaa1010		
	ADR8 or DATA	pp	2	2-5
	ADR16	qq	3	4
PSH	ACX	0011011x	1	4
PUL	ACX	0011001x	1	4
ROL	ACX	01yy1001	1	2
	ADR8	pp	2	7
	ADR16	qq	3	8
ROR	ACX	01yy0110	1	2
	ADR8	pp	2	7
	ADR16	qq	3	6
RTI		3B	1	10
RTS		39	1	5
SBA		10	1	2
SBC	ACX,	1xaa0010		
	ADR8 or DATA	pp	2	2-5
	ADR16	qq	3	4
SEC		0D	1	2
SEI		0F	1	2
SEV		0B	1	2
STA	ACX,	1xaa0111	•	
	ADR8	pp	2	4-8
	ADR18	qq	3	5

Table 6-2. MC6800 Instruction Set Object Codes (Continued)

MNEMONIC	OPERAND(S)	OBJECT CODE	BYTE	MACHINE CYCLES
STS		10aa1111	*	
	ADR8	pp	2	5-7
	ADR16	qq	3	6
STX		11aa1111	*	
	ADR8	pp	2	5-7
	ADR16	qq	3	6
SUB	ACX,	1xaa0000		
	ADR8 or DATA	pp	2	2-5
	ADR16	qq	3	4
SWI		3F	1	12
TAB		16	1	2
TAP		06	1	2
TBA		17	1	2
TPA		07	1	2
TST	ACX	01yy1101	1	2
	ADR8	pp	2	7
	ADR16	qq	3	6
TSX		30	1	4
TXS		35	1	4
WAI		3E	1	9

*aa = 00 is not permitted.

MC6800 ADDRESSING MODES

The Motorola MC6800 offers seven basic addressing methods:

1) Memory — Immediate
2) Memory — Direct
3) Memory — Indexed
4) Memory — Extended
5) Inherent
6) Relative
7) Accumulator

MC6800 instructions allow various combinations of these addressing modes to address the operands required for the instruction. See Table 6-3 for the addressing options available with each instruction.

MEMORY — IMMEDIATE

In this form of addressing, one of the operands is present in the byte(s) immediately following the first byte of object code. An immediate operand is specified by prefacing the operand with the # symbol. For example:

        ADD A   #$30

requests the Assembler to generate an ADD instruction which will add the value 30_{16} to Accumulator A.
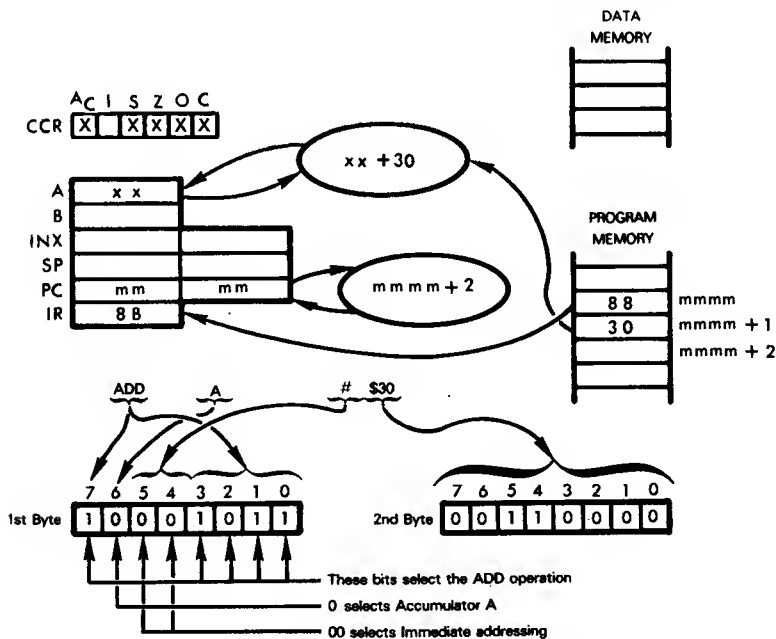
This example demonstrates a single byte immediate operand. Instructions such as AND, BIT, EOR and SBC use this form.

Double-byte immediate operands are employed by the CPX, LDS and LDX instructions. The LDS instruction, for example, may load the Stack Pointer with the two bytes following the first object code byte. The instruction:

LDS   #$3F2A

is illustrated in the following diagram:
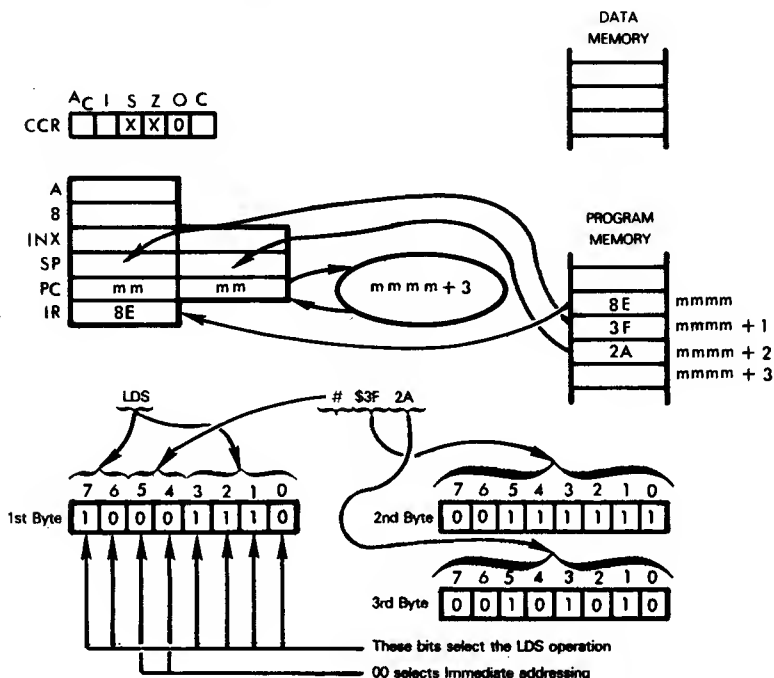
This instruction stores the contents of memory location mmmm + 1 into the high order byte of the Stack Pointer, then stores the contents of memory location mmmm + 1 into the low order byte of the Stack Pointer.

MEMORY — DIRECT

This form of addressing uses the second byte of the instruction to identify an operand present in the low 256_{10} words of memory. This form of addressing is specified when the expression used as the operand reduces to a value between 00_{16} and FF_{16}. For example:

ADD A $30

requests the Assembler to generate an ADD instruction which will add the value present at memory location 0030_{16} to Accumulator A.
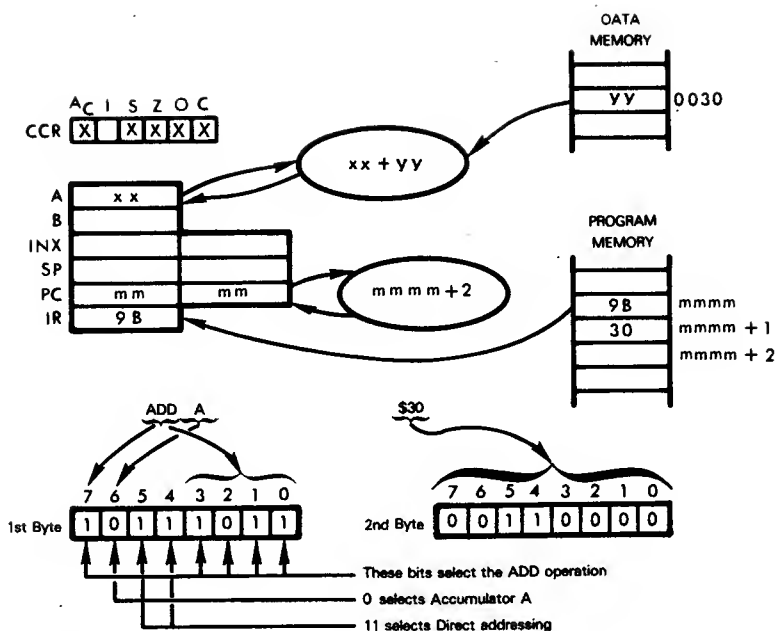
MEMORY — INDEXED

This form of addressing combines the second byte of the instruction with the contents of the Index register to produce the memory address of the data to be used as an operand. Indexed addressing on the MC6800 differs from indexed addressing as described in "An Introduction To Microcomputers: Volume I" in that the one-byte displacement provided by the memory reference instruction is added to the Index register as an unsigned 8-bit value. The contents of the Index register are not changed.
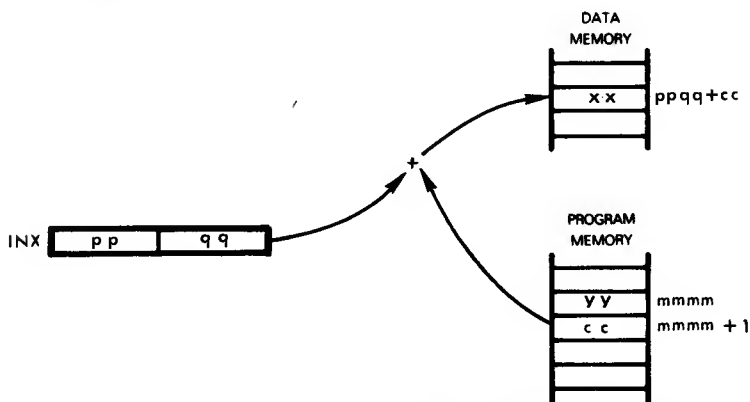
If instruction yy specifies indexed addressing, then the effective address of one operand will be ppqq + cc. Therefore xx will be one of the operands used by the instruction.

Indexed addressing is specified by including:

$$X$$
$$,X$$
$$expr,X$$

in the operand field of the instruction. For example:

ADD A $30,X

requests the Assembler to generate an ADD instruction which will add to Accumulator A the value present at the memory location specified by adding 30_{16} to the contents of the Index register. If the Index register contains 0316_{16} at the time this instruction is executed, the following diagram summarizes the instruction execution:
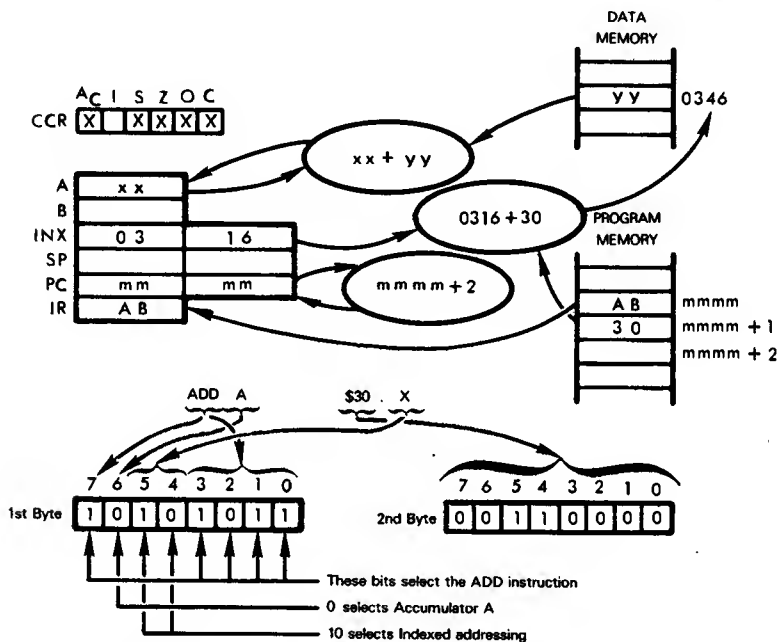
This instruction adds the contents of Accumulator A to the contents of the memory location specified by adding the second byte of the instruction to the contents of the Index register.

Note that implied addressing may be implemented by specifying indexed addressing with a displacement of 0.

MEMORY — EXTENDED

This form of addressing combines the second and third bytes of object code to form the address of the data to be used as an operand. Motorola extended addressing is identical to the direct addressing mode described in "An Introduction To Microcomputers: Volume I". Extended addressing is specified in the same way as direct addressing, i.e., an expression is given as the operand. If the expression evaluates to a number in the range $256 \leq$ expression $\leq 65,355$ then extended

addressing is used. For example:

    ADD A $31F6

requests the Assembler to generate an ADD instruction which will add the value present at memory location 31F6₁₆ to the contents of Accumulator A.
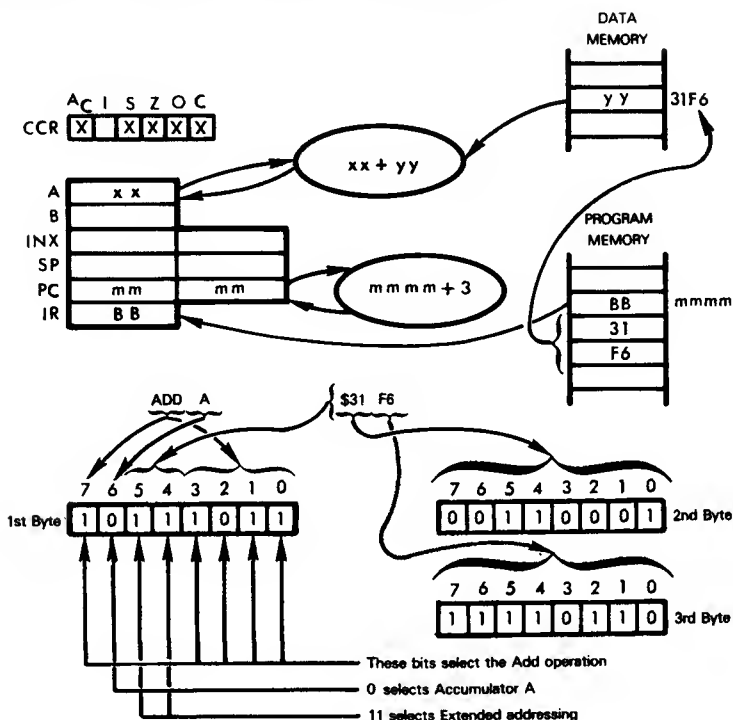
In addition, there are several instructions which do not provide a direct addressing option $(0 \leq expr \leq 255)$, e.g., CLR, DEC, ROR. For those instructions, extended addressing is used whenever a memory location is to be directly accessed.

INHERENT

Inherent addressing is specified when it is obvious by the nature of the instruction mnemonic which registers, statuses or memory locations are to be used as operands. For example, ABA, the Add Accumulator B to Accumulator A instruction, specifies what registers are to be the operands. CLI, the Clear Interrupt Mask instruction, specifies what status is to be affected by the instruction and RTS, the Return from Subroutine instruction, specifies that a return is to be executed; this will access the stack to determine the new value of the Program Counter.

RELATIVE

Branch and Branch-on-Condition instructions use program relative addressing; a single byte displacement is treated as a signed binary number which is added to the Program Counter, after the Program Counter contents have been incremented to address the next sequential instruction. This allows displacements in the range + 129₁₀ to -125₁₀ bytes.

ACCUMULATOR

Accumulator addressing is used by instructions having a single operand. Most of these instructions can select either Accumulator A or Accumulator B or a memory byte via the operand. For example, the

        CLR A

instruction is one of four forms of the CLR instruction:

1) CLR A — Clear Accumulator A
2) CLR B — Clear Accumulator B
3) CLR expr,X — Clear memory byte selected by indexed addressing
4) CLR expr — Clear memory byte selected by extended addressing

The CLR A instruction is illustrated in the following diagram:

Table 6-3. Addressing Options

	IMMEDIATE	DIRECT	INDEXED	EXTENDED	INHERENT	RELATIVE	ACCUMULA-TOR
ABA					X		
ADC*	X	X	X	X			
ADD*	X	X	X	X			
AND*	X	X	X	X			
ASL			X	X			X
ASR			X	X			X
BCC						X	
BCS						X	
BEQ						X	
BGE						X	
BGT						X	
BHI						X	
BIT*	X	X	X	X			
BLE						X	
BLS						X	
BLT						X	
BMI						X	
BNE						X	
BPL						X	
BRA						X	
BSR						X	
BVC						X	
BVS						X	
CBA					X		
CLC					X		
CLI					X		
CLR			X	X			X
CLV					X		
CMP*	X	X	X	X			
COM			X	X			X
CPX	X	X	X	X			
DAA					X		
DEC			X	X			X
DES					X		
DEX					X		
EOR*	X	X	X	X			
INC			X	X			X
INS					X		
INX					X		
JMP			X	X			
JSR			X	X			
LDA*	X	X	X	X			
LDS	X	X	X	X			
LDX	X	X	X	X			
LSR			X	X			X
NEG			X	X			X
NOP					X		
ORA*	X	X	X	X			
PSH							X
PUL							X
ROL			X	X			X
ROR			X	X			X
RTI					X		
RTS					X		
SBA					X		
SBC*	X	X	X	X			
SEC					X		

Table 6-3. Addressing Options (Continued)

	IMMEDIATE	DIRECT	INDEXED	EXTENDED	INHERENT	RELATIVE	ACCUMULATOR
SEI					X		
SEV					X		
STA		X	X	X			
STS		X	X	X			
STX		X	X	X			
SUB*	X	X	X	X			
SWI					X		
TAB					X		
TAP					X		
TBA					X		
TPA					X		
TST			X	X			X
TSX					X		
TXS					X		
WAI					X		

*These are dual operand instructions; one operand is from memory and the other is the selected Accumulator.

ABA — ADD ACCUMULATOR B TO ACCUMULATOR A
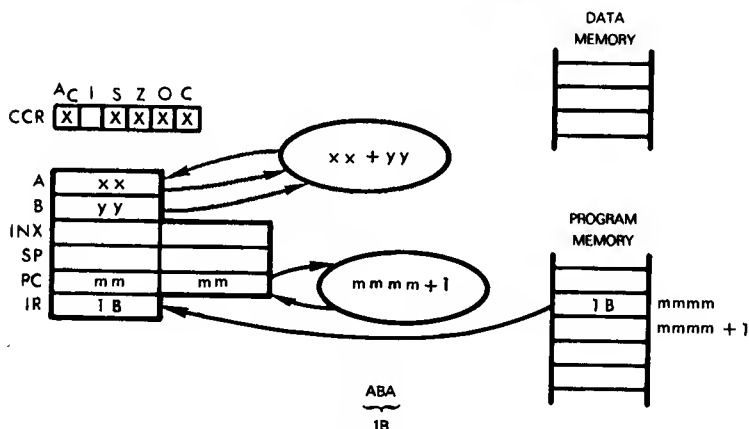
Add the contents of Accumulator B to the contents of Accumulator A. Store the result in Accumulator A. If $xx = B4_{16}$ and $yy = 2D_{16}$, then after the instruction:

    ABA

has executed, Accumulator A will contain $E1_{16}$.

$$B4 = 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0$$
$$2D = 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1$$
$$1\ 1\ 1\ 0\ 0\ 0\ 0\ 1$$

No Carry, set C to 0

Nonzero result, set Z to 0

There is a carry out of bit 3, set A_C to 1

Bit 7 is 1, set S to 1

0 is set to 0 since the XOR of the carry out of bits 6 and 7 is 0
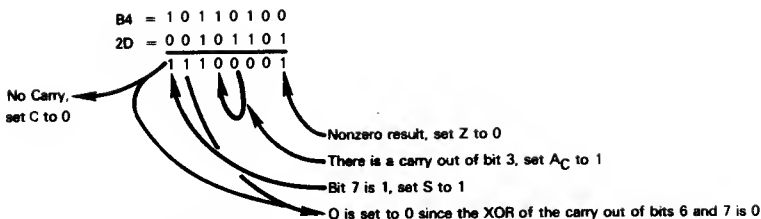
This is a routine data manipulation instruction.

ADC — ADD MEMORY, WITH CARRY, TO ACCUMULATOR A OR B

This instruction uses four methods of addressing data memory and allows the contents of data memory and the carry status to be added to Accumulator A or B. The four methods of addressing memory are:

1) Immediate
2) Direct
3) Extended
4) Indexed

The first byte of object code determines which addressing options are selected:

ADC

A
B

[#expr]
[expr]
[expr,X]

7 6 5 4 3 2 1 0

| 1 | y | x | x | 1 | 0 | 0 | 1 |

This bit selects the Accumulator
0 Accumulator A
1 Accumulator B

These two bits select the memory addressing mode
00 Immediate addressing (2 object bytes)
01 Direct addressing (2 object bytes)
10 Indexed addressing (2 object bytes)
11 Extended addressing (3 object bytes)
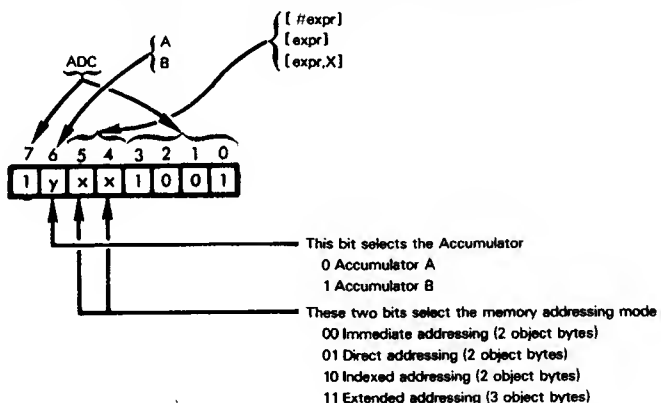
First, consider performing an addition with carry using immediate data.
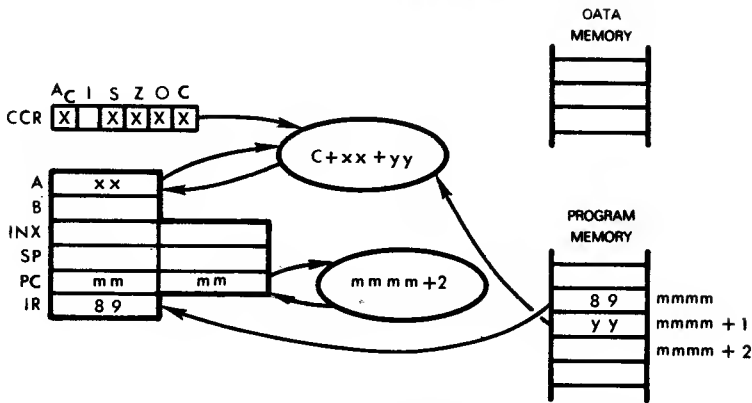
To Accumulator A, (selected by bit 6 of the byte in the Instruction register), add the contents of the next program memory byte, (addressing mode selected by bits 5 and 4 of the byte in the Instruction register) and the Carry status. Suppose $xx = 3A_{16}$, $yy = 7C_{16}$, $C = 1$. After the instruction:

ADC A   #$7C

has executed, the Accumulator will contain $B7_{16}$:
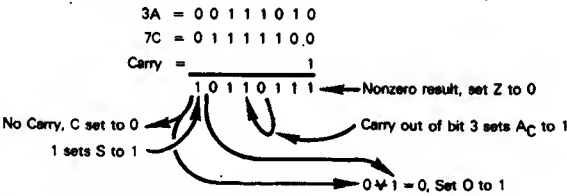
Consider adding using direct memory addressing:
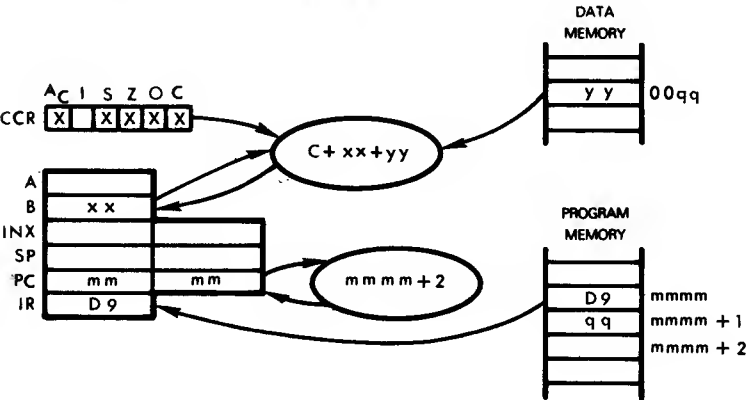
To Accumulator B add the Carry status and the contents of the data memory addressed by the next program memory byte (mmmm + 1). Note that the next program memory byte will address data memory bytes in the range $0_{10} \leq qq \leq 255_{10}$. If $xx = 3A_{16}$, $qq = 1F_{16}$, $yy = 7C_{16}$ and $C = 1$, then execution of the instruction:

ADC B $1F

generates the same result as execution of the previously described ADC A   #$7C instruction, with the exception that the result is stored in Accumulator B instead of Accumulator A.

Addition with carry using extended addressing works in a similar manner to direct addressing:
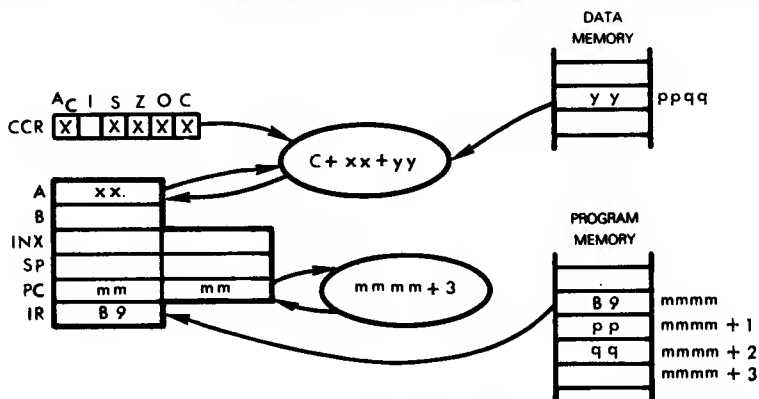
To Accumulator A, add the Carry status and the contents of data memory addressed by the next two program memory bytes, (high order address byte in the second object code byte, mmmm + 1, and the low order address byte in the third object code byte, mmmm + 2). Note that the two program bytes can address data memory bytes in the range $0 \leq ppqq \leq 65,355_{10}$. If $xx = 3A_{16}$, $pp = 50_{16}$, $qq = 23_{16}$, $yy = 7C_{16}$ and $C = 1$, then execution of the instruction:

ADC A $5023

produces the same result as execution of the ADC A   #$7C instruction which was described above.

Indexed addressing takes two different forms. Indexed addressing with no displacement, similar to implied addressing described in "An Introduction To Microcomputers: Volume I" uses the contents of the Index register to ascertain the memory address to be referenced.
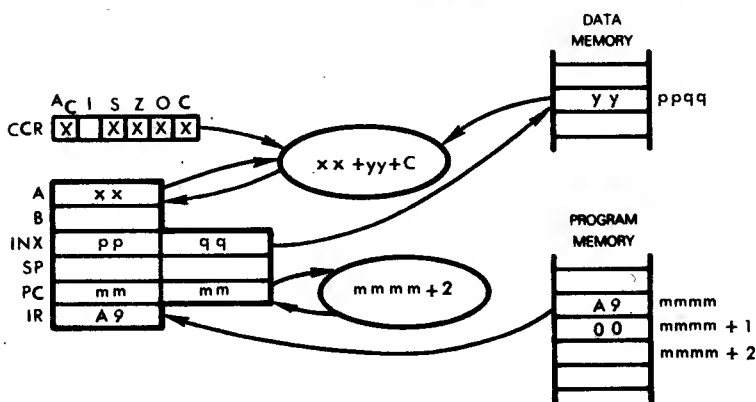
To Accumulator A, add the Carry status and the contents of data memory addressed by the Index register. Note that the Index register can address data memory bytes in the range $0 \leq ppqq \leq 65,355$. If $xx = 3A_{16}$, ppqq (the contents of INX) $= 5023_{16}$, yy $= 76_{16}$ and $C = 1$, then execution of the instruction:

     ADC A   X

produces the same result as ADC A $5023 which has been described above.

Indexed addressing with displacement allows a displacement in the byte following the instruction to be added to the contents of the Index register.
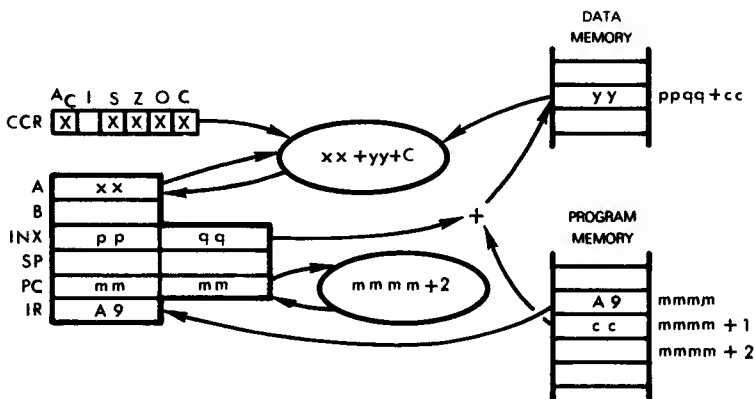
To Accumulator A, add the contents of memory addressed by the sum of the Index register and the program memory byte following the instruction code. (Note that in the previous example mmmm + 1 was 0), and the Carry status. The value in mmmm + 1 is treated as an 8-bit unsigned integer when the addition with the Index register is performed. If $xx = 3A_{16}$, $ppqq = 500D_{16}$, $cc = 16_{16}$, yy $= 76_{16}$ and $C = 1$, then the instruction:

     ADC A $16,X

generates the same result as the ADC A $5023 instruction discussed previously.

The ADC instruction is most frequently used in multibyte additions, to include the carry in the addition of the second and subsequent bytes.

ADD — ADD MEMORY TO ACCUMULATOR

This instruction ADDs the contents of a memory location to Accumulator A or B. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using direct addressing; consult the ADC instruction for the other available modes.
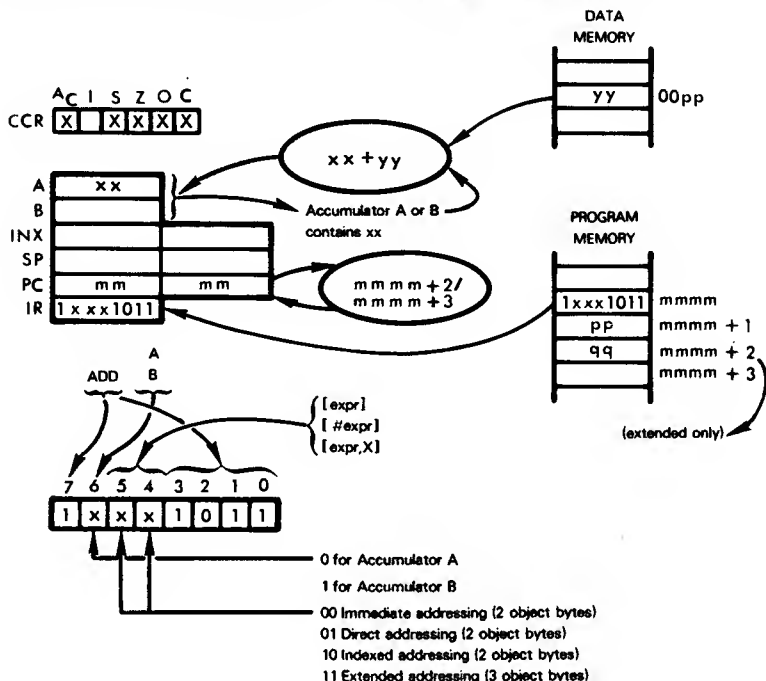
To the selected Accumulator, add the contents of the selected memory byte. Suppose $xx = 24_{16}$, $yy = 8B_{16}$, $pp = 43_{16}$ and $C = 1$. After the instruction:

    ADD A $43

has executed, Accumulator A will contain AF_{16}:
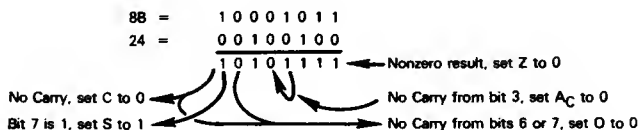
ADD is the binary addition instruction used in normal, single-byte operations; it is also the instruction used to add the low order bytes of two multibyte numbers.

AND — AND MEMORY WITH ACCUMULATOR

This instruction ANDs the contents of a memory location with the contents of Accumulator A or B. This instruction offers the same memory addressing options as the ADC instruction, and will be

illustrated using immediate addressing; consult the description of the ADC instruction for the other addressing modes.
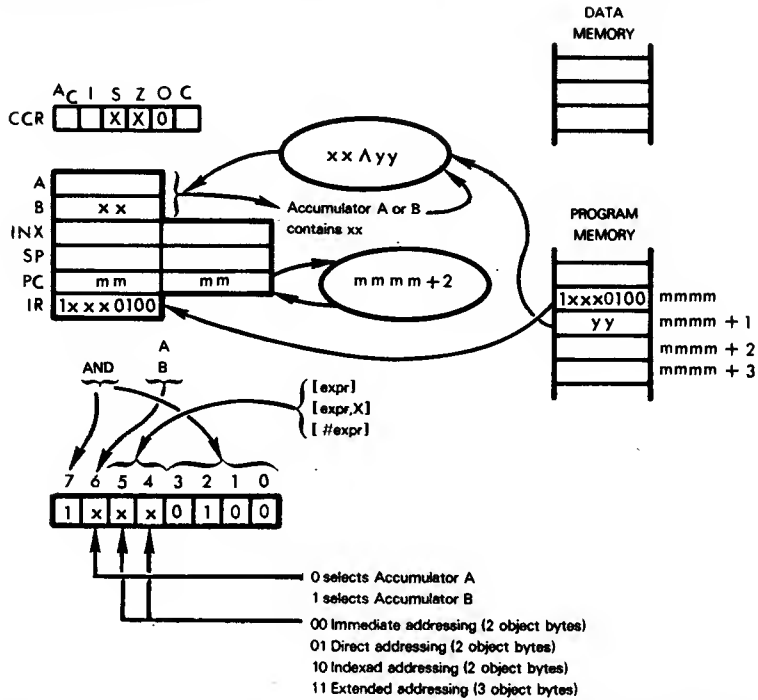
AND the contents of the selected memory byte with the selected Accumulator and store the result in the selected Accumulator. Suppose $xx = FC_{16}$ and $yy = 13_{16}$. After the instruction:

```
AND B    #$13
```

has executed, Accumulator B will contain 10_{16}:

```
FC =      1 1 1 1 1 1 0 0
13 =      0 0 0 1 0 0 1 1
          0 0 0 1 0 0 0 0 ◄── Nonzero result. Set Z to 0
```

0 in bit 7 sets S to 0   0 is cleared to 0
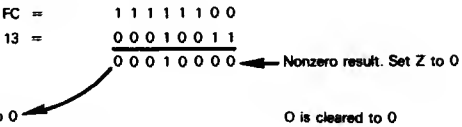
AND is a frequently used logical instruction.

ASL — SHIFT ACCUMULATOR OR MEMORY BYTE LEFT

Perform a one-bit arithmetic left shift of the contents of Accumulator A or B or the contents of the selected memory byte.

First, consider shifting an Accumulator:

```
                                              OATA
                                              MEMORY

        A C I S Z O C
CCR  [  | |X|X|X|X|]

      A  [        ]        0                  PROGRAM
      B  [        ]                           MEMORY
    INX  [        ]
     SP  [        ]
     PC  [  m m  |  m m  ]   ( m m m m + 1 )
     IR  [010 x 1000]                      [010×1000]  mmmm
                                                       mmmm + 1

              A
      ASL     B

      7  6  5  4  3  2  1  0
    [ 0| 1| 0| x| 1| 0| 0| 0 ]

                        0 selects Accumulator A
                        1 selects Accumulator B
```
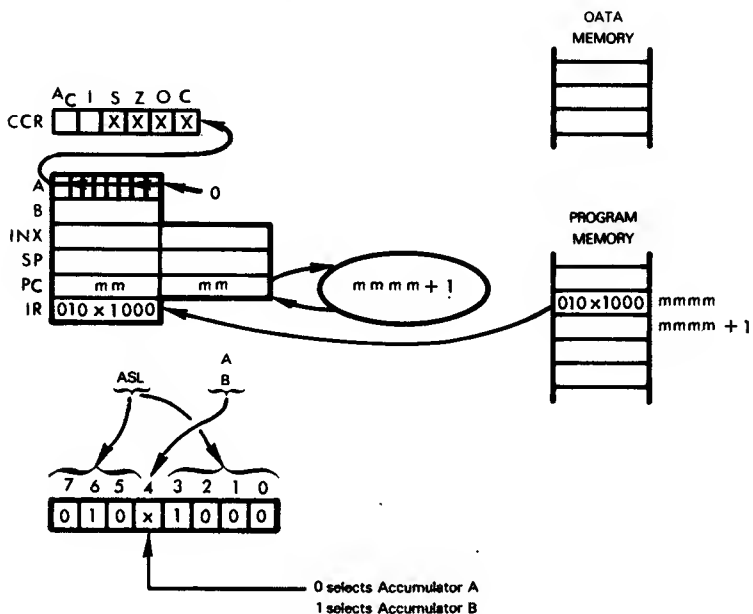
Suppose Accumulator A contains $7A_{16}$. Performing an:

    ASL A

instruction will set the Carry status to 0, the Sign status to 1, the Overflow status to 1, the Zero status to 0 and store $F4_{16}$ in Accumulator A.

```
            Carry     Accumulator A
             X        0 1 1 1 1 0 1 0
   ASL A     0        1 1 1 1 0 1 0 0  ───── Nonzero result, set Z to 0

  Set S to 1  ◄──                 ──►  0 ¥ 1 = 1, set O to 1
```
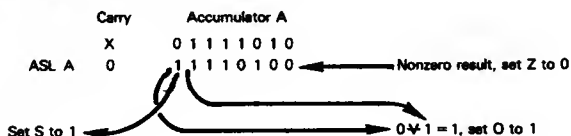
The ASL instruction uses two data memory addressing options:

1) Extended
2) Indexed
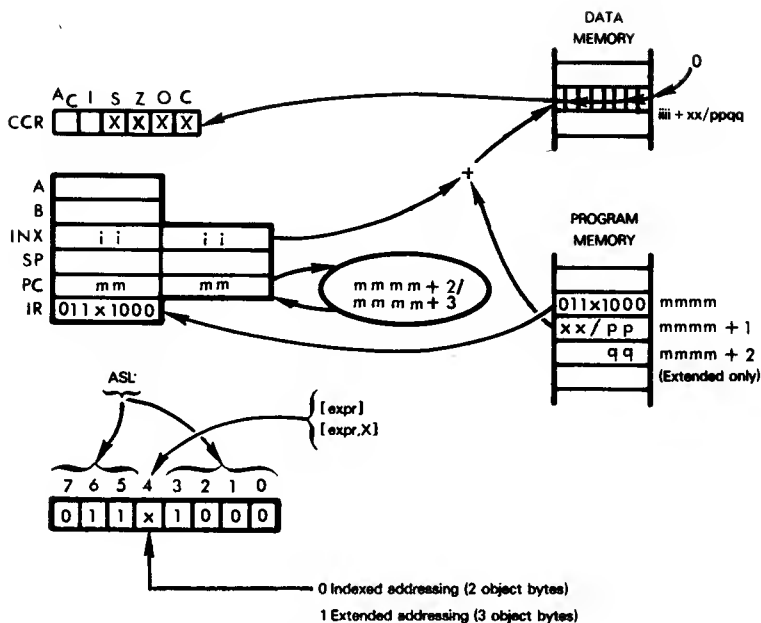
Suppose indexed addressing is used, iiii = $3F3C_{16}$, xx = $4A_{16}$, ppqq = $3F86_{16}$ and the contents of ppqq are CB_{16}. After executing an:

ASL $4A,X

instruction, the contents of ppqq will be altered to 96_{16} and Carry will be set to I:
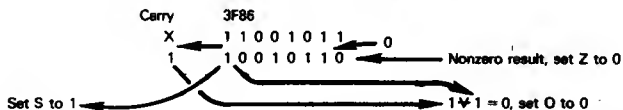
The ASL instruction is often used in multiplication routines and as a standard logical instruction. Note that a single ASL instruction multiplies its operand by 2.

ASR — SHIFT ACCUMULATOR OR MEMORY BYTE RIGHT

Perform a one-bit arithmetic right shift of the contents of Accumulator A or B or the contents of a selected memory byte.

First, consider right shifting an Accumulator:
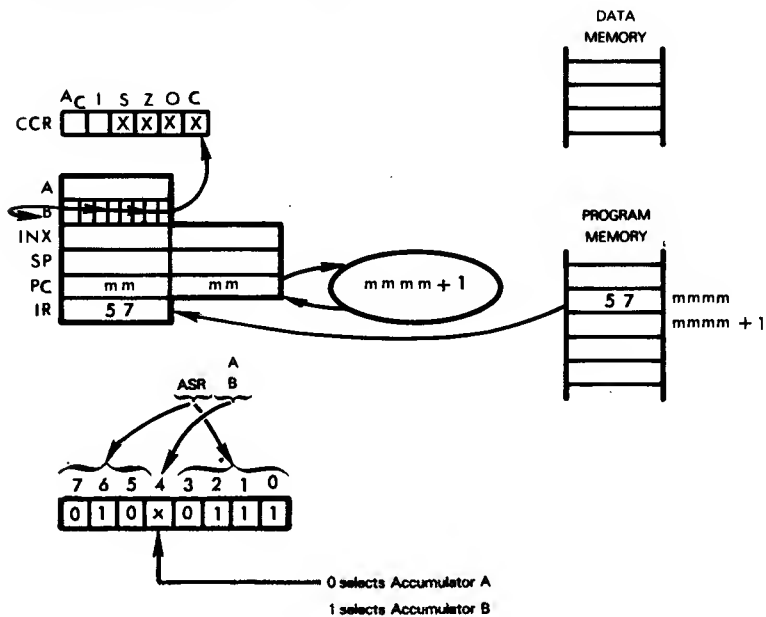
Suppose Accumulator B contains $7A_{16}$. Performing an:

ASR B

instruction will set Carry to 0, Sign to 0, Overflow to 0, Zero to 0 and store $3D_{16}$ in Accumulator B.
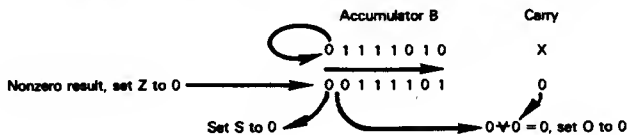
The ASR instruction uses two memory addressing options:

1) Extended
2) Indexed
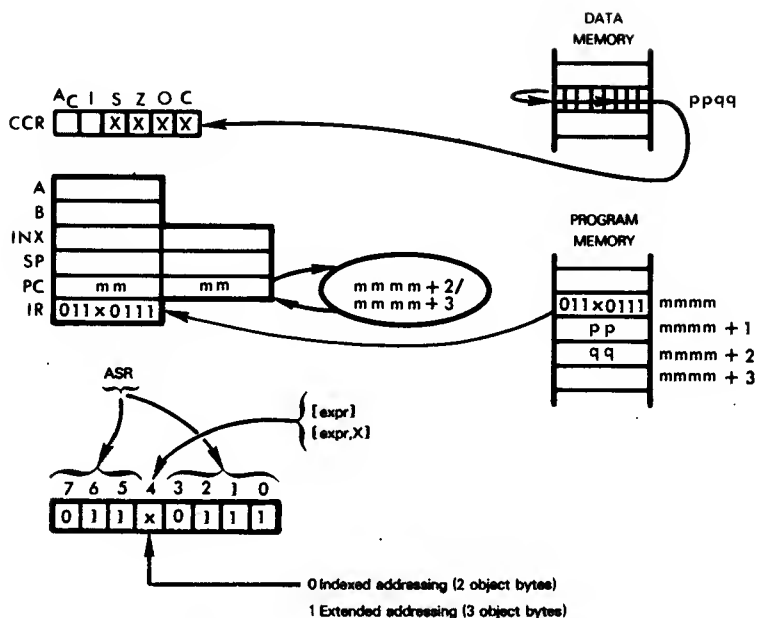
Suppose extended addressing is used, pp = 01_{16}, qq = 34_{16}, and the contents of 0134_{16} are CB_{16}. Executing an:

         ASR $0134

instruction will alter the contents of memory location 0134_{16} to $E5_{16}$.
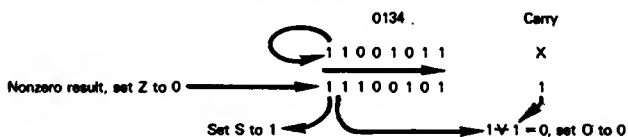
ASR is frequently used in division routines.

BCC — BRANCH IF CARRY CLEAR

$$\underbrace{BCC}_{24}$$

This instruction is identical to the BRA instruction except that the branch is only executed if the Carry status equals 0, otherwise the next instruction is executed.

In the following instruction sequence:
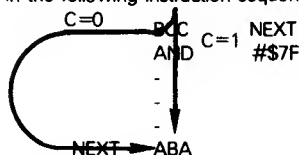
after the BCC instruction, the ABA instruction is executed if the Carry status equals 0. The AND instruction is executed if the Carry status equals 1.

BCS — BRANCH IF CARRY SET

BCS
︵
25

This instruction is identical to the BRA instruction except that the branch is only executed if the Carry status equals 1, otherwise the next instruction is executed.

In the following instruction sequence:
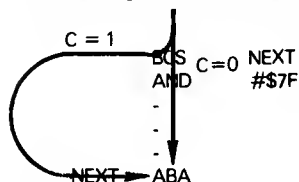
After the BCS instruction, the ABA instruction is executed if 'the Carry status equals 1. The AND instruction is executed if the Carry status equals 0.

BEQ — BRANCH IF EQUAL

BEQ
︵
27

This instruction is identical to the BRA instruction except that the branch is executed only if the Zero status equals 1; otherwise the next instruction is executed.

In the following instruction sequence:
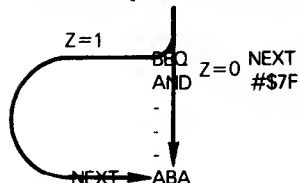
After the BEQ instruction, the ABA instruction is executed if the Zero status equals 1. The AND instruction is executed if the Zero status equals 0.

BGE — BRANCH IF GREATER THAN OR EQUAL TO ZERO

BGE
︵
2C

This instruction is identical to the BRA instruction except that the branch is executed only if the Exclusive-OR of the Sign and Overflow statuses is 0; i.e., Sign and Overflow are both 1 or Sign and Overflow are both 0; otherwise the next instruction is executed.

In the following instruction sequence:

```
(S ∀ O)=0
         ┌──BGE   (S ∀ O)=1   NEXT
         │  AND                #$7F
         │  -
         │  -
         │  -
         └──NEXT──►ABA
```
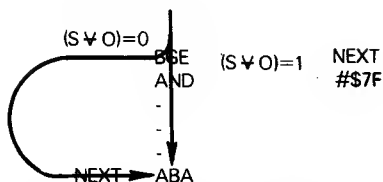
After the BGE instruction, the ABA instruction is executed if the Sign and Overflow statuses are both 1 or if they are both 0. The AND instruction is executed if the Sign status does not equal the Overflow status.

This instruction is used to perform a twos complement greater than or equal branch.

BGT — BRANCH IF GREATER THAN ZERO

BGT
‿
2E

This instruction is identical to the BRA instruction except that the branch is executed only if one of the following two conditions is met:

1) The Zero status is 0 and the Sign and Overflow statuses are 0.

2) The Zero status is 0 and the Sign and Overflow statuses are 1.

Otherwise, the next instruction is executed.

In the following instruction sequence:

```
Z V (S ∀ O) = 0
         ┌──BGT   Z V (S ∀ O) =1   NEXT
         │  AND                     #$7F
         │  -
         │  -
         │  -
         └──NEXT──►ABA
```
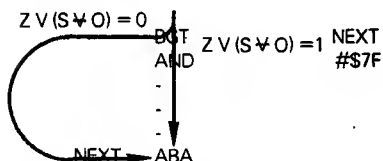
After the BGT instruction, the ABA instruction is executed if the Zero flag is 0 and the Exclusive-OR of the Sign and Overflow statuses is 0. In all other cases, the AND instruction is executed.

This instruction is used to implement a twos complement greater than branch capability.

BHI — BRANCH IF HIGHER

BHI
‿
22

This instruction is identical to the BRA instruction except that the branch is executed only if the Zero status and the Carry status are 0; otherwise the next instruction is executed.

In the following instruction sequence:

```
Z V C = 0
         ┌──BHI   Z V C = 1   NEXT
         │  AND                #$7F
         │  -
         │  -
         │  -
         └──NEXT──►ABA
```
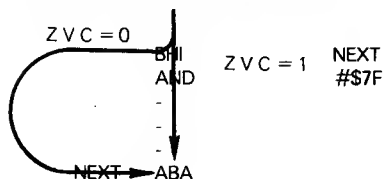
After the BHI instruction is executed, the ABA instruction is executed if the Zero and Carry statuses are 0. The AND instruction is executed if either the Zero or Carry status is 1 or both the Zero and Carry statuses are 1.

This instruction provides an unsigned greater than branch instruction. Contrast this instruction with the BGT instruction.

BIT — BIT TEST

This instruction ANDs the contents of Accumulator A or B with the contents of a selected memory location, sets the condition flags accordingly, but does not alter the contents of the Accumulator or memory byte. This instruction offers the same memory addressing options as the ADC instruction. This instruction will be illustrated using extended addressing; consult the ADC instruction for the other available modes.
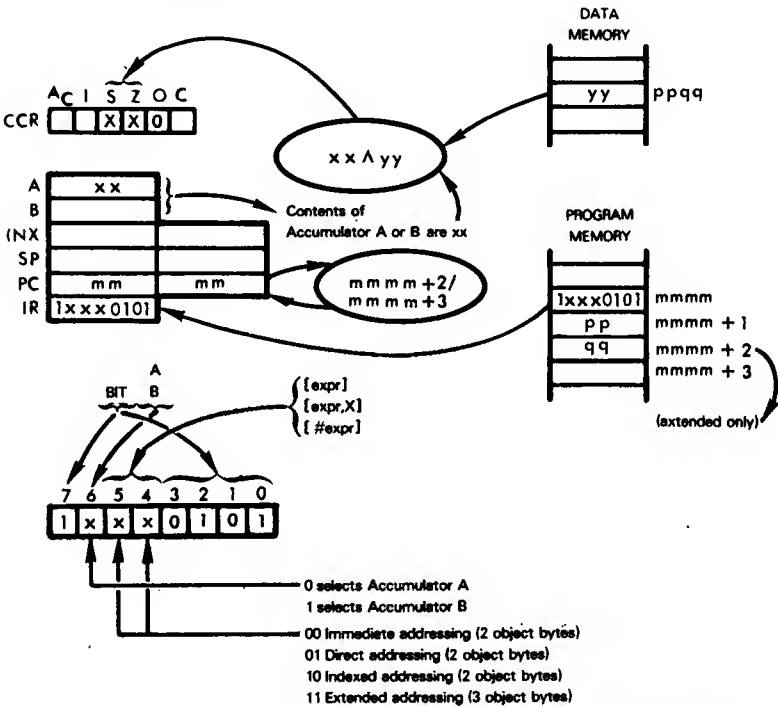
AND the contents of the specified Accumulator with the contents of the selected memory location and set the Sign and Zero condition flags accordingly. Suppose $xx=A6_{16}$, $yy=E0_{16}$ and $ppqq=1641_{16}$. After the instruction:

        BIT   A $1641

has executed, Accumulator A will still contain $A6_{16}$, location ppqq will still contain $E0_{16}$ but the statuses will be modified as follows:

BIT instructions frequently precede conditional Branch instructions. BIT instructions are also used to perform masking functions on data.

BLE — BRANCH IF LESS THAN OR EQUAL TO ZERO

BLE
~~~
2F

This instruction is identical to the BRA instruction with the exception that the branch is executed only if one or more of the three following conditions exist:

1) The Zero status is 1.
2) The Overflow status is 1 and the Sign status is 0.
3) The Overflow status is 0 and the Sign status is 1.

Otherwise the next instruction is executed.

In the following instruction sequence:

```
Z V (S ∀ O) = 1        BLE      Z V (S ∀ O) = 0   NEXT
                       AND                        #$7F
                        -
                        -
                        -
         NEXT    ABA
```

After the BLE instruction, the ABA instruction is executed if the Zero status is 1 or the Exclusive-OR of the Sign and Overflow statuses is 1. The AND instruction is executed if the Zero status is 0 and the Exclusive-OR of the Sign and Overflow statuses is 0.

This instruction provides the programmer with a twos complement less than or equal branch.

# BLS — BRANCH IF LOWER OR SAME

BLS
~~~
23
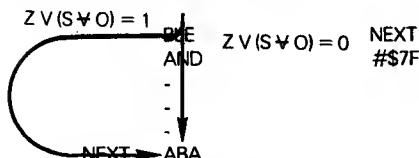
This instruction is identical to the BRA instruction except that the branch is executed only if either the Carry or Zero status is set; otherwise the next instruction is executed.
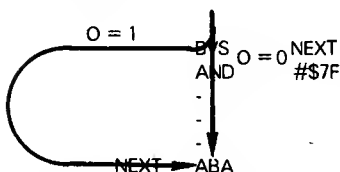
In the following instruction sequence:

```
C V Z = 1        BLS      C V Z = 0     NEXT
                 AND                    #$7F
                  -
                  -
                  -
      NEXT    ABA
```
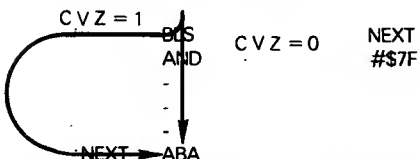
After the BLS instruction, the ABA instruction is executed if the Carry or Zero status equals 1. The AND instruction is executed if both the Carry and Zero statuses are 0.

This instruction is useful as an unsigned less than or equal branch. Compare this instruction with the BLE instruction which provides a twos complement less than or equal branch.

BLT — BRANCH IF LESS THAN ZERO
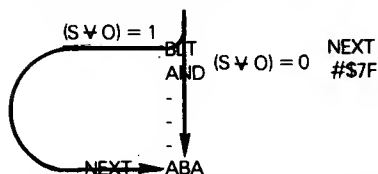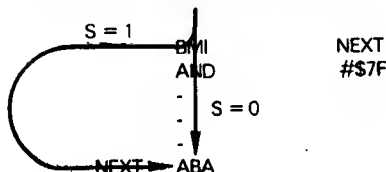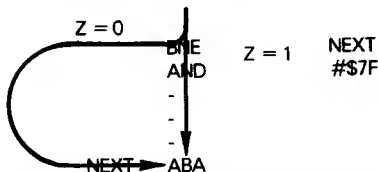
BLT
~~~
2D

This instruction is identical to the BRA instruction except that the branch is performed only when

the Exclusive-OR of the Sign and Overflow statuses is 1; otherwise the next instruction is executed.

In the following instruction sequence:

After the BLT instruction, the ABA instruction is executed if the Sign and Overflow statuses are not equal. The AND instruction will be executed if the Sign and Overflow statuses are equal.

This instruction provides a twos complement less than branch capability.
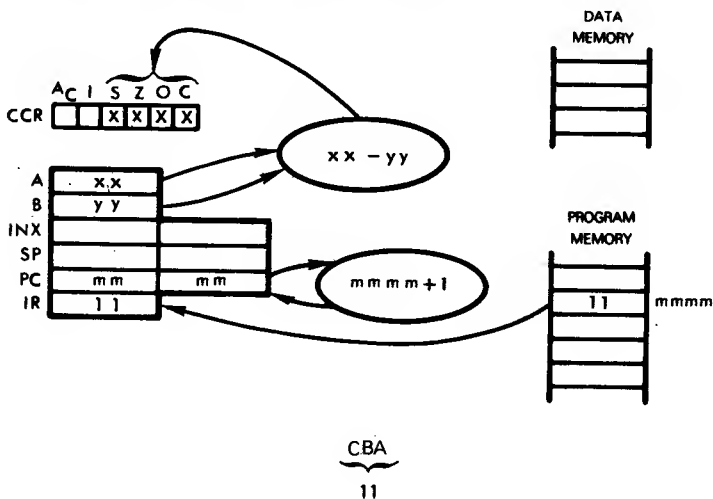
BMI — BRANCH IF MINUS

BMI
2B

This instruction is identical to the BRA instruction except that the branch is executed only if the Sign status is 1; otherwise the next instruction is executed.

In the following instruction sequence:

After the BMI instruction, the ABA instruction is executed if the Sign status is 1. The AND instruction is executed if the Sign status is 0.

BNE — BRANCH IF NOT EQUAL

BNE
26

This instruction is identical to the BRA instruction except that the branch is executed only if the Zero status is 0; otherwise the next instruction is executed.

In the following instruction sequence:

After the BNE instruction, the ABA instruction is executed if the Zero status is 0. The AND instruction is executed if the Zero status is 1.

BPL — BRANCH IF PLUS

BPL
$\overbrace{}$
2A

This instruction is identical to the BRA instruction except that the branch is executed only if the Sign status is 0; otherwise the next instruction is executed.

In the following instruction sequence:

```
S = 0          BPL           NEXT
               AND   S = 1   #$7F
                -
                -
                -
        NEXT    ABA
```

After the BPL instruction, the ABA instruction is executed if the Sign status is 0. The AND instruction is executed if the Sign status is 1.

BRA — BRANCH TO THE INSTRUCTION IDENTIFIED IN THE OPERAND

BRA   disp
$\overbrace{}$ $\overbrace{}$
20   rr

This instruction adds the contents of the second object code byte (taken as a signed 8-bit displacement) to the contents of the Program Counter plus 2; this becomes the memory address for the next instruction to be executed. The previous Program Counter contents are lost.

In the following instruction sequence:

```
        BRA     NEXT
        AND     #$7F
         -
         -
         -
NEXT    ABA
```

After the BRA instruction, the ABA instruction will be executed. The AND instruction will never be executed unless a Branch or Jump instruction somewhere else in the instruction sequence jumps to this instruction. (Note that since this instruction is not labeled, this is a very unlikely event.)

6-40

The Branch instruction uses Branch-Relative addressing, which is similar to Program Relative Paging as described in "An Introduction To Microcomputers: Volume I — Basic Concepts". The exception is that the Program Counter contents are incremented to point to the next instruction before the 8-bit signed displacement is added. Therefore, the Program Counter contents are replaced by:

$$[PC] + 2 + rr$$

Note that in the above example, the ABA instruction labeled by NEXT must be within 129 object bytes (not instructions) of the Branch-to-Next instruction. If it isn't, the Assembler will flag the BRA instruction as an error.

BSR — BRANCH TO THE SUBROUTINE IDENTIFIED IN THE OPERAND

Store the address of the instruction following the BSR on the top of the stack; the top of the stack is a data memory byte addressed by the Stack Pointer. Then subtract two from the Stack Pointer in order to address the new top of stack. Add the contents of the second byte of the instruction and two to the Program Counter and begin execution.

Consider the instruction sequence:

```
        BSR     SUBR
        AND     #$7F
        -
        -
        -
SUBR    ABA
```

After the BSR instruction has executed, the address of the AND instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The ABA instruction will be executed next.

BVC — BRANCH IF OVERFLOW CLEAR

BVC
‿‿
28

This instruction is identical to the BRA instruction except that the branch is executed only if the Overflow status is 0; otherwise the next instruction is executed.

In the following instruction sequence:

After the BVC instruction, the ABA instruction is executed if the Overflow status is 0. The AND instruction is executed if the Overflow status is 1.

BVS — BRANCH IF OVERFLOW SET

$$\underbrace{BVS}_{29}$$

This instruction is identical to the BRA instruction except that the branch is executed only if the Overflow status is 1; otherwise the next instruction is executed.

In the following instruction sequence:

after the BVS instruction, the ABA instruction is executed if the Overflow status equals 1. The AND instruction is executed if the Overflow status equals 0.

CBA — COMPARE ACCUMULATORS

$$\underbrace{CBA}_{11}$$

Subtract the contents of Accumulator B from the contents of Accumulator A. Discard the result, i.e., do not affect the contents of either Accumulator, but modify the status flags to reflect the result of the operation.

Suppose $xx = E3_{16}$ and $yy = A0_{16}$. After the instruction:

        CBA

has executed, Accumulator A will still contain $E3_{16}$ and Accumulator B will still contain $A0_{16}$, but statuses will be modified as follows:

```
                    E3    =   1 1 1 0 0 0 1 1
   Twos complement of A0  =   0 1 1 0 0 0 0 0
                             0 1 0 0 0 0 1 1 ◄── Nonzero result, set Z to 0

     Carry sets C to 0
        0 sets S to 0 ──────────────────► 1 ∨ 1 ── set O to 0
```

Notice that the resulting Carry is complemented.

Compare instructions usually precede conditional Branch instructions.

CLC — CLEAR CARRY


```
          CLC
          ‾‾‾
          0C
```

Clear the Carry status. No other status or register's contents are affected.

The Carry status is also cleared by the CLR and TST instructions.

CLI — CLEAR INTERRUPT MASK

Clear the interrupt mask bit in the Condition Code register. This instruction enables the MC6800's interrupt service ability, i.e., the MC6800 will respond to the Interrupt Request control line. No other registers or statuses are affected.

CLR — CLEAR ACCUMULATOR OR MEMORY

This instruction clears a specified Accumulator or a selected memory byte. The Zero flag is set to 1; the Sign, Overflow and Carry statuses are set to 0.

First, consider clearing an Accumulator:

Clear the contents of the specified Accumulator. Suppose Accumulator B contains 43_{16}. After the instruction:

        CLR B

is executed, Accumulator B will contain 00_{16}. In addition, Sign, Overflow and Carry will be 0; Zero will be 1.

The CLR instruction also has two memory addressing modes, Indexed and Extended.

Suppose $pp = 43_{16}$, $qq = 14_{16}$ and $xx = 05_{16}$. After the execution:

        CLR $4314

instruction, the contents of memory location 4314_{16} will be 00 and the status flags will be appropriately modified.

CLV — CLEAR OVERFLOW

Clear the overflow bit in the Condition Code register. No other registers or statuses are affected.

CMP — COMPARE ACCUMULATOR WITH MEMORY

This instruction subtracts the contents of a selected memory byte from Accumulator A or B, sets the condition flags accordingly, but does not alter the contents of the Accumulator or memory byte. This instruction offers the same memory addressing options as the ADC instruction. This instruction will be illustrated using indexed addressing; consult the ADC instruction for examples of the other available modes.

DATA MEMORY

CCR

A_C I S Z O C
X X X X

xx − yy

yy   ppqq+cc

A   xx
B
INX   pp   qq
SP
PC   mm   mm
IR   1xxx0001

Accumulator A or B contains xx

PROGRAM MEMORY

mmmm+2 /
mmmm+3

1xxx0001   mmmm
cc   mmmm + 1
mmmm + 2
(extended only)

CMP   A   B

[expr]
[expr,X]
[#expr]

7 6 5 4 3 2 1 0
1 x x x 0 0 0 1

0 selects Accumulator A
1 selects Accumulator B

00 Immediate addressing (2 object bytes)
01 Direct addressing (2 object bytes)
10 Indexed addressing (2 object bytes)
11 Extended addressing (3 object bytes)

Subtract the contents of the selected memory byte from the contents of the specified Accumulator and set the Sign, Zero, Overflow and Carry statuses to reflect the result of the subtraction. Suppose $xx = F6_{16}$, $yy = 18_{16}$ and $cc = 43_{16}$.

After the instruction:

    CMP B $43,X

has executed, Accumulator B will still contain $F6_{16}$, location ppqq + 43_{16} will still contain 18_{16} but the statuses will be modified as follows:

```
              F6  =  1 1 1 1 0 1 1 0
Twos complement of 18  ≈  1 1 1 0 1 0 0 0
                  1 1 0 1 1 1 1 0 ◄──── Nonzero result, set Z to 0
   Set C to 0
   Set S to 1                      1 ⊕ 1 = 0, set O to 0
```

Notice that C is the complement of the resulting carry.

Compare instructions are most frequently used to set statuses before the execution of Branch-on-Condition instructions.

COM — COMPLEMENT ACCUMULATOR OR MEMORY

This instruction complements a specified Accumulator or a selected memory byte.

First, consider complementing an Accumulator:

Complement the contents of the specified Accumulator. No other status bit or register's contents are affected. Suppose Accumulator B contains $3A_{16}$. After the instruction:

> COM B

is executed, the Accumulator will contain $C5_{16}$.

$$3A_{16} = 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0$$
$$\text{Complement} = 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1 \longleftarrow \text{Nonzero result, Z is set to 0}$$
Carry is set to 1
S is set to 1
Overflow is set to 0

The COM instruction also offers two memory addressing modes:

1) Extended
2) Indexed

Suppose that the contents of the Index register are 0100_{16}, and the contents of memory location 0113_{16} are 23_{16}. After a:

     COM $13,X

instruction executes, memory location 0113_{16} will be altered to DC_{16}.

CPX — COMPARE INDEX REGISTER

This instruction compares the contents of the Index register with the contents of two selected memory locations. This instruction offers the same memory addressing options as the ADC instruction. This instruction will be illustrated using direct addressing; consult the discussion of the ADC instruction for the other addressing modes.

DATA
MEMORY

```
AC I S Z O C
CCR [  [X]X]X]  ]

        A
        B
INX   P P    q q
 SP
 PC   m m    m m
 IR  10x x1100
```

pp — x x
qq — y y

x x   00rr
y y   00rr + 1

PROGRAM
MEMORY

mmmm+2/
mmmm+3

10x x1100   mmmm
r r   mmmm + 1
s s   mmmm + 2
            (extended and
            immediate)

CPX

{ [#expr]
  [expr,X]
  [expr] }

```
7 6 5 4 3 2 1 0
[1 0 x x 1 1 0 0]
```

00 Immediate addressing (3 object bytes)
01 Direct addressing (2 object bytes)
10 Indexed addressing (2 object bytes)
11 Extended addressing (3 object bytes)

Subtract the contents of the memory byte immediately following the selected memory byte from the low byte of the Index register, and discard the result. Subtract the contents of the selected memory byte from the high byte of the Index register. Set the Sign and Overflow statuses according to the result, set the Zero status to 1 if the result of both subtractions was 0, and discard the result.

Suppose $pp = 1A_{16}$, $qq = B0_{16}$, $xx = 1B_{16}$, $yy = B0_{16}$ and $rr = 43_{16}$. After the instruction:

    CPX $43

has executed, the Index register will still contain $1AB0_{16}$, location 0043_{16} will still contain $1B_{16}$ and memory location 0044_{16} will still contain $B0_{16}$, but the statuses Sign, Zero and Overflow will be modified as follows:

```
                1A   =   0 0 0 1 1 0 1 0
Twos complement of 1B =   1 1 1 0 0 1 0 1
                        1 1 1 1 1 1 1 1    Both results are not zero, set Z to 0
        Sets S to 1
                                           0 ∀ 0 = 0, set O to 0

                B0   =   1 0 1 1 0 0 0 0
Twos complement of B0 =   0 1 0 1 0 0 0 0
                          0 0 0 0 0 0 0 0
```

DAA — DECIMAL ADJUST ACCUMULATOR

DATA MEMORY

PROGRAM MEMORY

A C I S Z O C
CCR [ ][ ]X X X X

A   x x
B
INX
SP
PC   m m   m m
IR   1 9

Convert to BCD

m m m m + 1

1 9   mmmm

DAA
19

Convert the contents of Accumulator A to binary-coded-decimal form. This instruction should be used only after adding two BCD numbers, i.e., look upon ABA DAA or ADD A DAA or ADC A DAA or SUB A DAA or SBC A DAA as compound decimal arithmetic instructions which operate on BCD source to generate BCD answers.

Suppose Accumulator A contains 39_{16} and Accumulator B contains 47_{16}. After the instructions:

    ABA
    DAA

have executed, the Accumulator will contain 86_{16}, not 80_{16}.

The Sign and Zero flags are modified to reflect the status they represent. The Overflow status is destroyed and the Carry status is set or reset as if a hypothetical binary-coded-decimal addition had just taken place.

DEC — DECREMENT ACCUMULATOR OR MEMORY

This instruction decrements a specified Accumulator or a selected memory byte.
Consider decrementing Accumulator A or B.

Subtract 1 from the contents of the specified Accumulator.

Suppose Accumulator B contains $3A_{16}$. After instruction:

     DEC B

has executed, Accumulator B will contain 39_{16}:

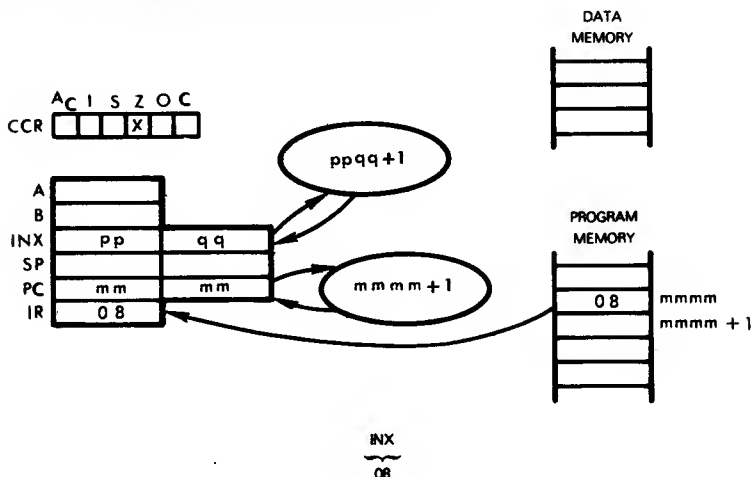The DEC instruction also offers two memory addressing modes:

1) Extended
2) Indexed

Decrement the contents of the specified memory byte.

If $xx = A5_{16}$, $ppqq = 0100_{16}$, and $cc = 0A_{16}$, then after execution of the instruction:

      DEC $0A,X

memory location $010A_{16}$ will be altered to $A4_{16}$.

```
            A5  =  1 0 1 0 0 1 0 1
Ones complement of 1  =  1 1 1 1 1 1 1 1
                       1 0 1 0 0 1 0 0  ◄── Nonzero result sets Z to 0
    Carry is not altered
       Sets S to 1  ◄───              1 ⩝ 1 sets O to 0
```

DES — DECREMENT STACK POINTER

DES
‿‿
34

Subtract 1 from the 16-bit value in the Stack Pointer. No other registers or condition codes are affected.

Suppose the Stack Pointer contains 2F7A₁₆. After the instruction:

    DES

has executed, the Stack Pointer will contain 2F79₁₆.
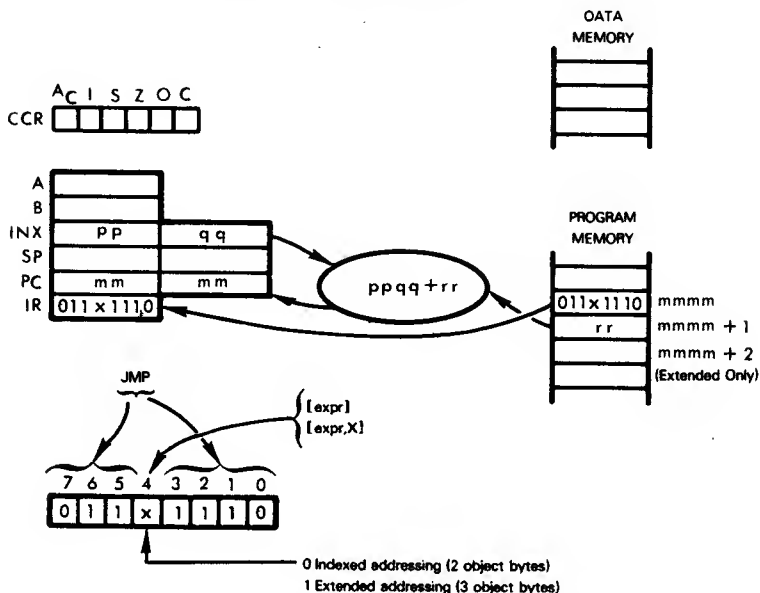
DEX — DECREMENT INDEX REGISTER

DEX
‿‿
09

Subtract 1 from the 16-bit value in the Index register. The Zero status is set to 1 if the 16-bit result is 0.

Suppose the Index register contains $310C_{16}$. After the instruction:

    DEX

has executed, the Index register will contain $310B_{16}$ and the Zero status will be set to 0.

EOR — EXCLUSIVE-OR ACCUMULATOR WITH MEMORY

Exclusive-OR the contents of Accumulator A or B with the contents of a selected memory byte. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using immediate addressing; consult the ADC instruction for the other addressing modes.

Exclusive-OR the contents of the specified Accumulator with the contents of the selected memory location, treating both operands as simple binary data. Suppose that $xx = E3_{16}$ and $yy = A0_{16}$. After the instruction:

    EOR A   #$A0

has executed, Accumulator A will contain 43_{16}.

$$
\begin{array}{rl}
E3 = & 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1 \\
A0 = & 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0 \\
\hline
& 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1
\end{array}
$$

Carry is not affected ←——— Nonzero result, set Z to 0

0 sets S to 0   Overflow is cleared

EOR is used to test for changes in bit status.

INC — INCREMENT ACCUMULATOR OR MEMORY

This instruction increments the specified Accumulator or selected memory byte.

First, consider incrementing an Accumulator:

Add 1 to the selected Accumulator. Suppose that xx = $3A_{16}$. After the instruction:

    INC   A

has executed, Accumulator A will contain $3B_{16}$.

The INC instruction also offers two memory addressing modes:

1) Extended
2) Indexed

Increment the selected memory byte.

If $pp = 01_{16}$, $qq = A2_{16}$ and $xx = C0_{16}$, then after executing an:

      INC $01A2

instruction, the contents of memory location $01A2_{16}$ will be incremented to $C1_{16}$.

The INC instruction can be used to provide a counter in a variety of applications, e.g., counting the occurrences of an event or as an iterative counter which specifies the number of times a task is to be performed.

INS — INCREMENT STACK POINTER

Add 1 to the 16-bit value in the Stack Pointer. No other registers or condition codes are affected.

Suppose the Stack Pointer contains $2F7A_{16}$. After the instruction:

INS

has executed, the Stack Pointer will contain $2F7B_{16}$.

INX — INCREMENT INDEX REGISTER

Add 1 to the 16-bit value in the Index register. The Zero status is set to 1 if the 16-bit result is 0.

Suppose the Index register contains $310C_{16}$. After the instruction:

INX

has executed, the Index register will contain $310D_{16}$.

JMP — JUMP VIA INDEXED OR EXTENDED ADDRESSING

This instruction will be illustrated using indexed addressing.

Jump to the instruction specified by the operand by loading the address of the selected memory byte into the Program Counter.

In the following instruction sequence:

```
        LDX     #JPTBL
        JMP     2,X
        -
        -
        -
JPTBL   BRA     NEWDATA
        BRA     PROCESSDATA
        BRA     FLAGDATA
```

If the Index register contains the address of JPTBL, then the JMP instruction will perform an indexed jump relative to JPTBL. In this case, the instruction executed following the:

```
        JMP     2,X
```

instruction would be the BRA PROCESSDATA instruction.

More frequently, the JMP instruction uses the extended addressing mode. In this case, the second byte of the instruction is loaded into the high byte of the Program Counter, and the third byte of the instruction is loaded into the low byte of the Program Counter. Instruction execution continues from this address.

JSR — JUMP TO SUBROUTINE USING INDEXED OR EXTENDED ADDRESSING

This instruction will be illustrated using extended addressing.

The Program Counter is incremented by 3 (if extended addressing is used), or 2 (if indexed addressing is used), and then is pushed onto the stack. The Stack Pointer is adjusted to point to the next empty location in the stack. (These functions are detailed in the description of the BSR instruction.) The address of the selected memory byte is then stored into the Program Counter. Execution continues from this point.

Consider this instruction sequence:

```
      JSR     SUBR
      AND     #$7F
      -
      -
      -
SUBR  ABA
```

After the JSR instruction has executed, the address of the AND instruction will be saved at the top of the stack. The ABA instruction will be the next instruction executed.

LDA — LOAD ACCUMULATOR FROM MEMORY

Load the contents of the selected memory byte into the specified Accumulator. This instruction offers the same memory addressing options as the ADC instruction and will be illustrated using indexed addressing; consult the ADC instruction for the other addressing modes.

Load the contents of the selected memory byte into the specified Accumulator.

Suppose the Index register contains 0800_{16} and cc $= 43_{16}$. If memory location 0843_{16} contains AA_{16}, then after:

　　　LDA A　　$43,X

has executed, Accumulator A will contain AA_{16}.

LDS — LOAD STACK POINTER

Load the contents of two selected memory locations into the Stack Pointer. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using immediate addressing; consult the discussion of the ADC instruction for examples of the other addressing modes.

Load the contents of the selected memory byte into the high byte of the Stack Pointer. Load the contents of the memory byte immediately following the selected memory byte into the low byte of the Stack Pointer. Set the Sign status if the most significant bit of the Stack Pointer is set; set the Zero status if all 16 bits loaded are 0. Clear the Overflow flag.

Suppose $pp = 14_{16}$ and $qq = 00_{16}$. After executing a:

    LDS   #$1400

instruction, the Stack Pointer will contain 1400_{16}.

LDX — LOAD INDEX REGISTER

Load the contents of two selected memory locations into the Index register. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using direct addressing; consult the ADC instruction for the other addressing modes.

Load the contents of the selected memory byte into the high byte of the Index register. Load the contents of the memory byte immediately following the selected memory byte into the low byte of the Index register. Set the Sign status if the most significant bit of the Index register is set. Set the Zero status if all 16 bits of the Index register are set to 0. The Overflow status is cleared to 0.

Suppose that $rr = 90_{16}$, $pp = 01_{16}$ and $qq = 00_{16}$. Executing the:

    LDX $90

instruction will load 0100_{16} into the Index register.

LSR — LOGICAL RIGHT SHIFT OF ACCUMULATOR OR MEMORY

This instruction performs a one-bit logical right shift of the specified Accumulator or the selected memory byte.

Shift the selected Accumulator's contents right one bit. Shift the low order bit into the Carry status. Shift a 0 into the high order bit.

Suppose Accumulator B contains $7A_{16}$. After the:

        LSR B

instruction is executed, Accumulator B will contain 3D and the Carry status will be set to 0.

Two methods of memory addressing are available with the LSR instruction, Indexed and Extended.

Logically shift the contents of the selected memory location right by one bit.

Suppose $pp = 04_{16}$, $qq = FA_{16}$ and the contents of memory location $04FA_{16}$ are $0D_{16}$. After the instruction:

      LSR $04FA

is executed, the Carry status will be 1 and the contents of location $04FA_{16}$ will be 06_{16}.

NEG — NEGATE ACCUMULATOR OR MEMORY

This instruction negates the specified Accumulator or the selected memory byte by replacing the Accumulator or memory byte with the twos complement of its contents.

First, consider negating an Accumulator:

Negate the contents of the specified Accumulator by taking the ones complement of the contents of the specified Accumulator and adding one to the result.

Suppose Accumulator A contains $3A_{16}$. After the instruction:

    NEG A

is executed, Accumulator A will contain $C6_{16}$.

The NEG instruction also offers two memory addressing options:

1) Extended
2) Indexed

Suppose the contents of the Index register are 0100_{16}, $cc = 1D_{16}$ and memory location $011D_{16}$ contains AA_{16}. After the instruction:

NEG $1D,X

is executed, memory location $011D_{16}$ will be altered to 56_{16}.

```
             AA              = 1 0 1 0 1 0 1 0
Ones complement of AA        = 0 1 0 1 0 1 0 1
            + 1                    + 1
                             ―――――――――――――――――――
                             0 1 0 1 0 1 1 0 ◄── Nonzero result, set Z to 0
     No Carry, set C to 0 ◄―
          Set S to 0 ◄―
                                         0∀0 = 0, set O to 0
```

NOP — NO OPERATION

This is a one-byte instruction which performs no operation except that the Program Counter is incremented. This instruction is present for two reasons:

1) The NOP instruction allows you to give a label to an object program byte:

    HERE   NOP

2) To fine tune delay times. Each NOP instruction adds two cycles to a delay.

NOP is not a very useful or frequently used instruction.

ORA — OR ACCUMULATOR WITH MEMORY

This instruction ORs the contents of Accumulator A or B with the contents of the selected memory byte. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using extended addressing; consult the ADC instruction for examples of the other addressing modes.

OR the contents of the specified Accumulator with the contents of the selected memory location, treating both operands as simple binary data.

Suppose that $pp = 16_{16}$, $qq = 23_{16}$, $xx = E3_{16}$ and $yy = AB_{16}$. After the instruction:

    ORA A $1623

has executed, Accumulator A will contain EB_{16}.

This is a logical instruction; it is often used to turn bits "on". For example, the instruction:

ORA A   #$80

will unconditionally set the high order bit in Accumulator A to 1.

PSH — PUSH ACCUMULATOR ONTO STACK

Push the contents of the selected Accumulator onto the top of the stack. The Stack Pointer is then decremented by 1. No other registers or statuses are affected.
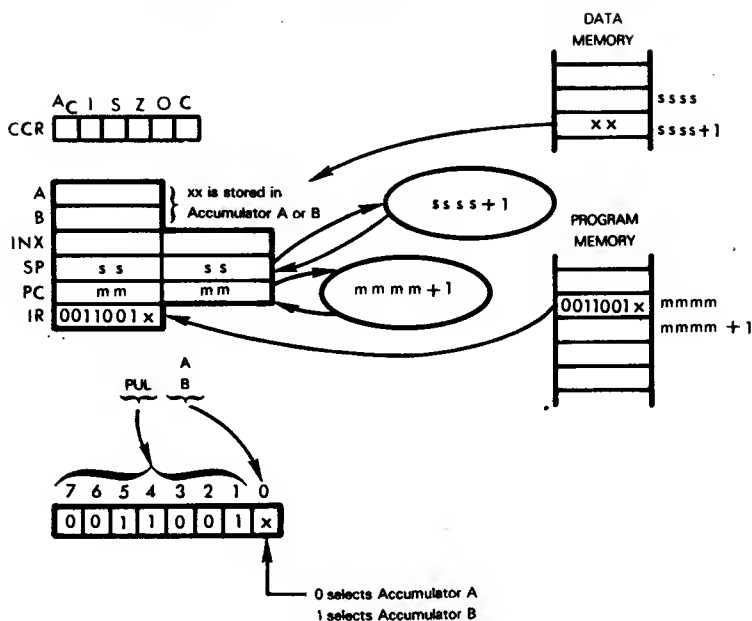
Suppose Accumulator A contains $3A_{16}$ and the Stack Pointer contains $2AF7_{16}$. After the instruction:

PSH A

has executed, $3A_{16}$ will have been stored into location $2AF7_{16}$ and the Stack Pointer will be altered to $2AF6_{16}$.

The PSH instruction is most frequently used to save Accumulator contents, for example, before servicing an interrupt.

PUL — PULL DATA FROM STACK

Increment the Stack Pointer, then pull the top stack byte into the selected Accumulator. No other registers or statuses are affected.

Suppose the Stack Pointer contains $2AF6_{16}$ and location $2AF7_{16}$ contains CE_{16}. After the instruction:
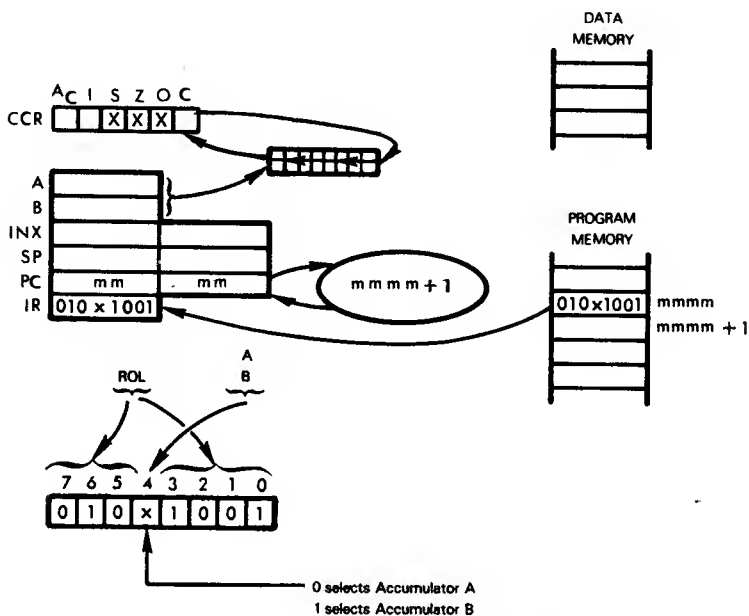
      PUL B

has executed, Accumulator B will contain CE_{16} and the Stack Pointer will contain $2AF7_{16}$.

The PUL instruction is most frequently used to restore Accumulator contents that have been saved on the stack, for example, after servicing an interrupt.

ROL — ROTATE ACCUMULATOR OR MEMORY LEFT THROUGH CARRY

This instruction rotates the specified Accumulator or the selected memory byte one bit to the left through the Carry.

First, consider rotating an Accumulator:

Rotate the selected Accumulator's contents left one bit through the Carry status.

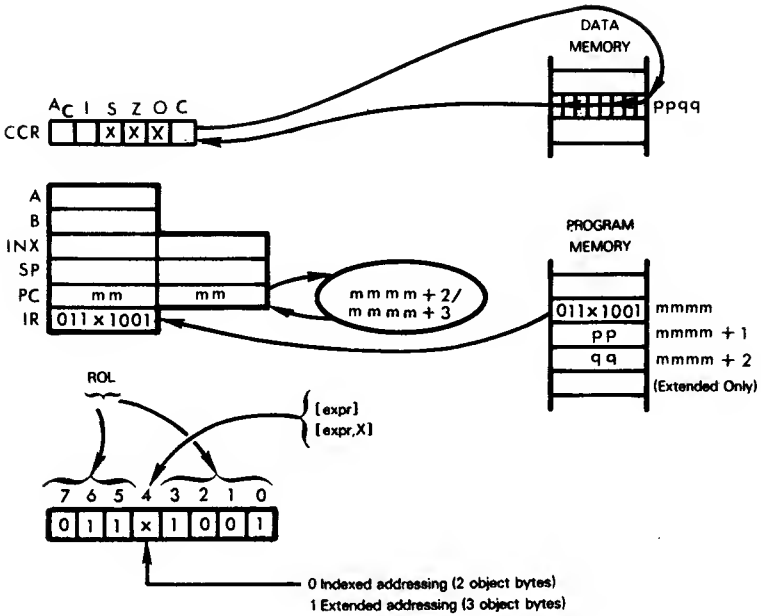Suppose Accumulator A contains $7A_{16}$ and the Carry status is set to 1. After the:

    ROL A

instruction is executed, Accumulator A will contain $F5_{16}$ and the Carry status will be reset to 0.

The ROL instruction provides two kinds of memory addressing:
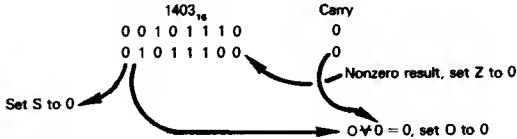
1) Extended
2) Indexed

Rotate the selected memory byte left one bit through the Carry status.

Suppose $pp = 14_{16}$, $qq = 03_{16}$, the contents of memory location 1403_{16} are $2E_{16}$ and the Carry status is 0. After executing a:

ROL $1403

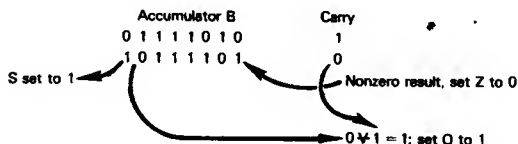instruction, memory location 1403_{16}'s contents will be $5C_{16}$.

ROR — ROTATE ACCUMULATOR OR MEMORY RIGHT THROUGH CARRY

This instruction rotates a specified Accumulator or a selected memory byte one bit to the right through the Carry.

First, consider rotating an Accumulator:

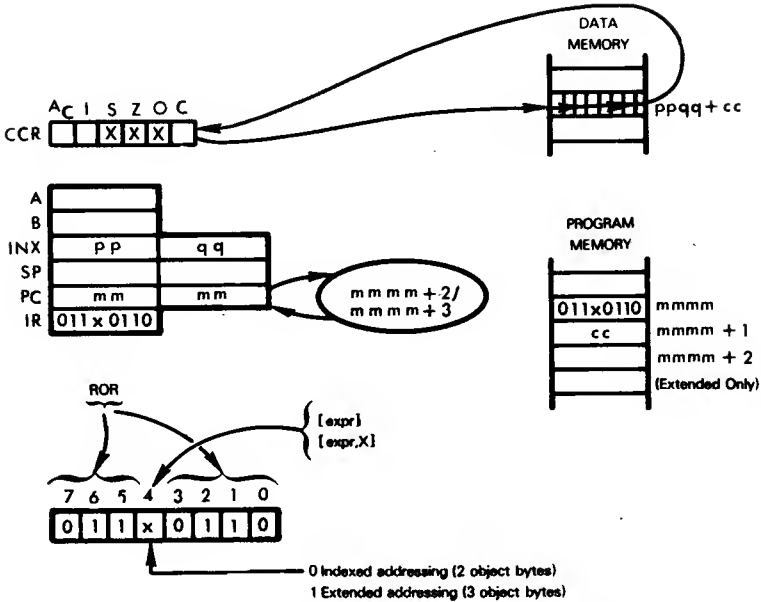Rotate the selected Accumulator's contents right one bit through the Carry status.

Suppose Accumulator B contains $7A_{16}$ and the Carry status is set to 1. Execution of the:

    ROR B

instruction will produce these results: Accumulator B will contain BD_{16} and the Carry status will be 0.

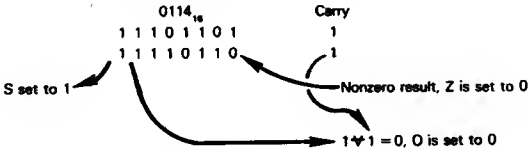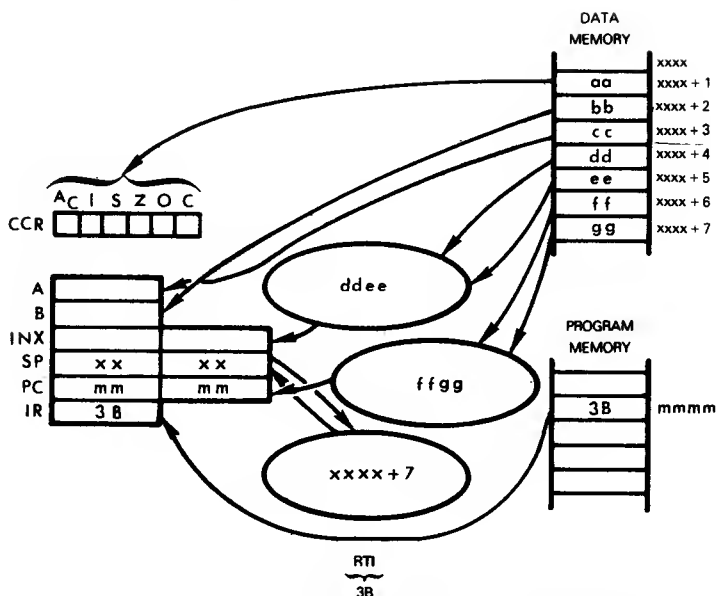The ROR instruction provides two kinds of memory addressing:

1) Extended

2) Indexed

Suppose that $cc = 14_{16}$, the contents of the Index register are 0100_{16}, the contents of memory location 0114_{16} are ED_{16} and the Carry status is 1. After executing a:

ROR $14,X

instruction, the Carry will be 1 and memory location 0114_{16} will contain $F6_{16}$.

RTI — RETURN FROM INTERRUPT

The Condition Code register, the Accumulators, the Index register and the Program Counter all have data values pulled into them off the stack. The registers and the corresponding locations on the stack which are pulled into the registers are as follows:

Memory Location (SP is xxxx at instruction execution start)	Register
xxxx + 1 (bits 5 - 0)	Condition Code register
xxxx + 2	Accumulator B
xxxx + 3	Accumulator A
xxxx + 4	High byte of Index register
xxxx + 5	Low byte of Index register
xxxx + 6	High byte of Program Counter
xxxx + 7	Low byte of Program Counter

Execution continues from the address pulled into the Program Counter.

Suppose the Stack Pointer contains $100F_{16}$, aa = CB_{16}, bb = 14_{16}, cc = 00_{16}, dd = 01_{16}, ee = 00_{16}, ff = 09_{16} and qq = $A2_{16}$. After the instruction:
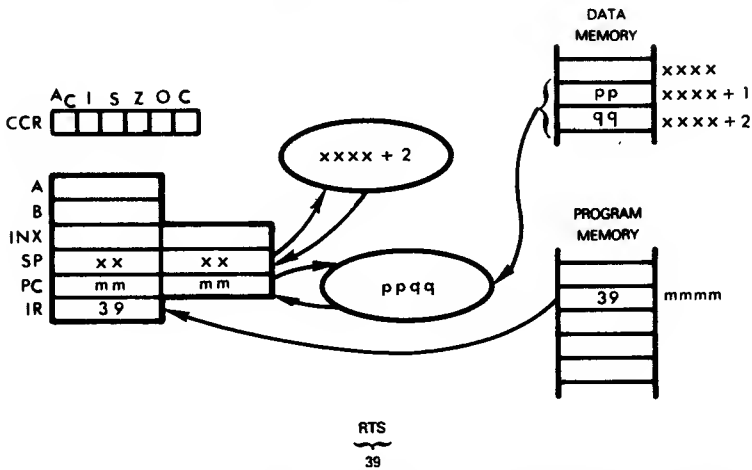
        RTI

has executed, Accumulator A will be 00_{16}, Accumulator B will contain 14_{16}, the Index register contents will equal 0100_{16}, the Stack Pointer will contain 1016_{16}, and the Program Counter contents will be $09A2_{16}$ (this is the address from which instruction execution will proceed). In addition, the Condition Code register will appear as follows:

Note that the Interrupt Mask bit will be set or reset depending on its value at the time the CCR was pushed.

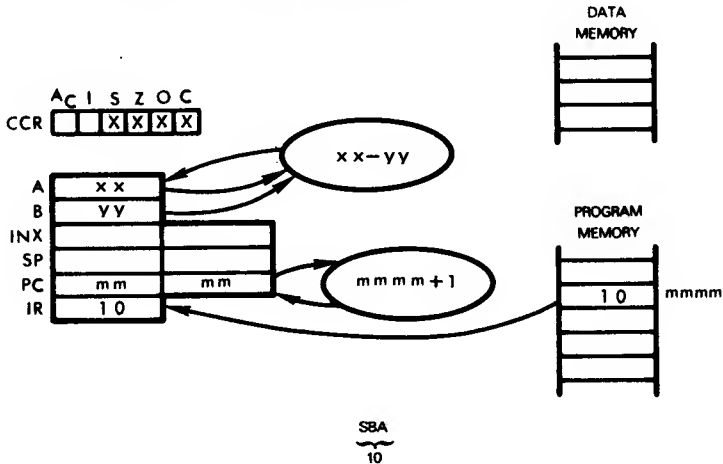# RTS — RETURN FROM SUBROUTINE

RTS
~~
39

Move the contents of the top two stack bytes to the Program Counter; these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2 to address the new top of stack.

Every subroutine must contain at least one Return instruction; this is the last instruction executed within the subroutine and causes execution to return to the calling program.

For an illustrated description of the RTS instruction's execution see Chapter 5.

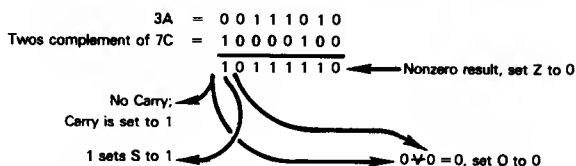# SBA — SUBTRACT ACCUMULATORS

SBA
~~
10

Subtract the contents of Accumulator B from the contents of Accumulator A.

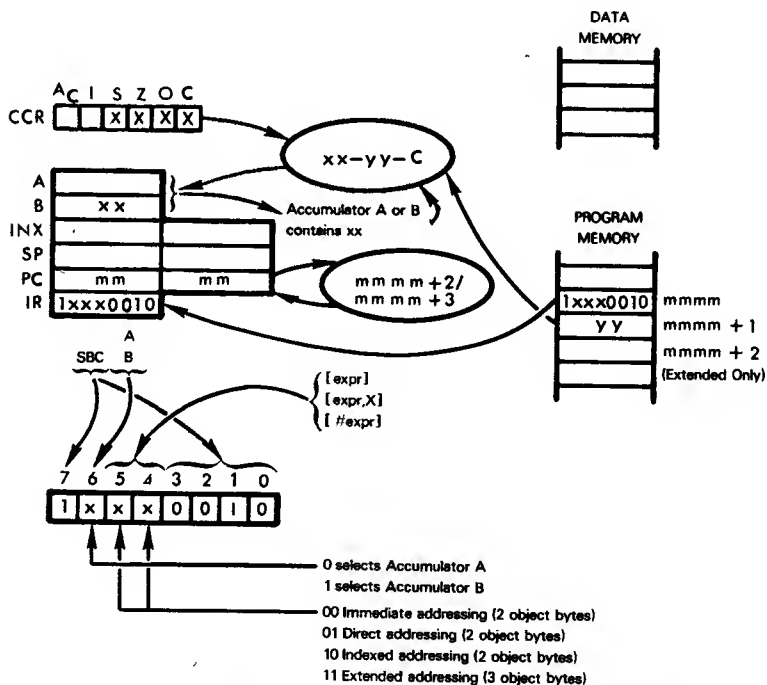Suppose $xx = 3A_{16}$ and $yy = 7C_{16}$. After the instruction:

    SBA

has executed, Accumulator A will contain BE_{16} and Accumulator B will contain $7C_{16}$.

Note that the resulting Carry is complemented.

SBC — SUBTRACT MEMORY FROM ACCUMULATOR WITH BORROW

Subtract the contents of the selected memory byte from the specified Accumulator. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using immediate addressing; consult the ADC instruction for examples of the other available modes.
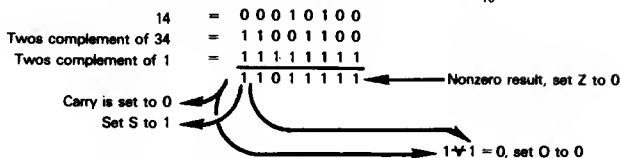

Subtract the contents of the selected memory byte, and the Carry status, from the specified Accumulator, treating all register contents as simple binary data.

Suppose $xx = 14_{16}$, $yy = 34_{16}$ and $C = 1$. After executing a:

    SBC B   #$34

instruction, the contents of Accumulator B would be altered to DF_{16}.

```
            14            =   0 0 0 1 0 1 0 0
Twos complement of 34     =   1 1 0 0 1 1 0 0
Twos complement of 1      =   1 1 1 1 1 1 1 1
                              1 1 0 1 1 1 1 1  ◄──── Nonzero result, set Z to 0

        Carry is set to 0 ◄──
            Set S to 1  ◄──
                              1 ⊻ 1 = 0, set O to 0
```
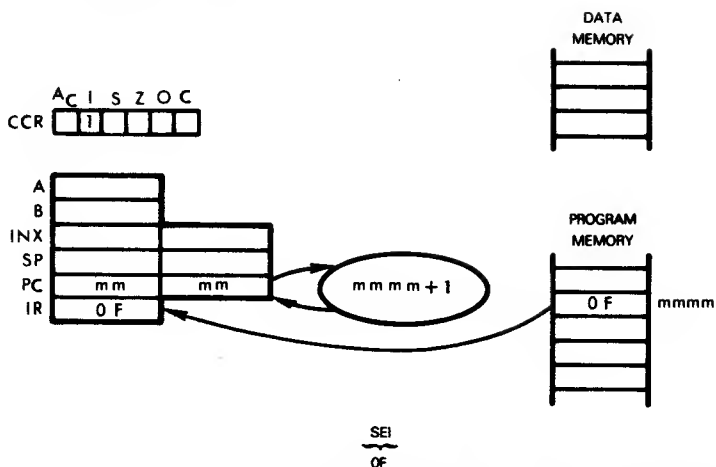
Note that the resulting Carry is complemented.

The SBC instruction is frequently used in multibyte subtraction, after the low order byte has been processed using the SUB instruction.

SEC — SET CARRY

When the SEC instruction is executed, the Carry status is set to 1, regardless of its previous value. No other statuses or register contents are affected.

SEI — SET INTERRUPT MASK

After this instruction has been executed, the microprocessor is inhibited from servicing an interrupt and will continue to execute instructions without responding to interrupts until the interrupt status is cleared. Non-maskable interrupts will be serviced regardless of the state of the Interrupt Mask bit.

With the exception of the Interrupt Mask bit in the CCR, no other registers or statuses are altered.

SEV — SET OVERFLOW STATUS

When the SEV instruction is executed, the Overflow status is set to 1, regardless of its previous value. This instruction does not affect any other statuses or register contents.

STA — STORE ACCUMULATOR IN MEMORY

Store the contents of the selected Accumulator into the specified memory location. This instruction offers the same memory addressing modes as the ADC instruction, with the exception that an immediate addressing mode is not available. This instruction will be illustrated using extended addressing; consult the ADC instruction for a discussion and example of indexed and direct addressing.

Store the specified Accumulator into memory.

Suppose $xx = 63_{16}$, $pp = 05_{16}$, $qq = 3A_{16}$. After the instruction:

STA   B $053A

is executed, the contents of memory location $053A_{16}$ will be 63_{16}.

STS — STORE STACK POINTER

Store the contents of the Stack Pointer into two contiguous memory locations. Like the STA instruction, this instruction offers direct, indexed and extended addressing modes. This instruction will be illustrated using direct addressing. Consult the ADC instruction for a discussion of indexed and extended addressing modes.

Store the high byte of the Stack Pointer into the selected memory byte. Store the low byte of the Stack Pointer into the memory byte immediately following the selected memory location.

Suppose the contents of the Stack Pointer are $28FF_{16}$ and rr = 80_{16}. After executing the:

STS

instruction, memory location 0080_{16} will contain 28_{16} and memory location 81_{16} will contain FF_{16}.

STX — STORE INDEX REGISTER

Store the contents of the Index register into two contiguous memory locations. Like the STA instruction, this instruction does not offer immediate addressing, but it does offer the other three memory access methods: Direct, Indexed and Extended. This instruction will be illustrated using extended addressing; consult the ADC instruction for a discussion of direct and indexed addressing.

Store the high byte of the Index register into the selected memory byte. Store the low byte of the Index register into the memory byte immediately following the selected memory location.

Suppose the contents of the Index register are 0100_{16}, pp = 14_{16}, and qq = 30_{16}. After the:

STX

instruction has executed, memory location 1430_{16} will contain 01_{16} and 1431_{16} will contain 00_{16}.

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 ◄—Nonzero result, set Z to 0

S is set to 0 ◄—   Overflow is cleared

SUB — SUBTRACT MEMORY FROM ACCUMULATOR

Subtract the contents of the selected memory byte from the contents of Accumulator A or B. This instruction offers the same memory addressing options as the ADC instruction, and will be illustrated using direct addressing; consult the description of the ADC instruction for examples of the other addressing modes.

Subtract the contents of the selected memory byte from the contents of the specified Accumulator, treating both operands as simple binary data.

Suppose $xx = E3_{16}$, $yy = A0_{16}$, and $rr = 31_{16}$. After executing the instruction:

        SUB B $31

the contents of Accumulator B will be 43_{16}.

The SUB instruction is used to perform single byte subtractions, or for the low order byte in multibyte subtractions.

SWI — SOFTWARE INTERRUPT

The Program Counter is incremented by one, then the Program Counter, Index register, Accumulators A and B, and the Condition Code register are all pushed onto the stack. The registers and the corresponding memory locations into which they are pushed are shown below:

Memory Location (SP is xxxx at start of instruction execution)	Register
xxxx	Low byte of Program Counter
xxxx-1	High byte of Program Counter
xxxx-2	Low byte of Index register
xxxx-3	High byte of Index register
xxxx-4	Accumulator A
xxxx-5	Accumulator B
xxxx-6	Condition Code register

The Interrupt Mask bit is then set to 1. This disables the MC6800's interrupt service ability, i.e., the processor will not respond to an interrupt from a peripheral device. The contents of the SWV (the Software Interrupt Pointer) are then loaded into the Program Counter.

The SWI instruction can be used for a variety of functions. The address of the entry point for a group of system subroutines or the address of the entry point for a disk operating system or the address of any software package could be inserted in the Software Interrupt Pointer. By executing an SWI instruction, any of these various software systems could be entered. For further information on the SWI instruction, consult Chapter 6 of "An Introduction To Microcomputers: Volume II — Some Real Products".

TAB — MOVE FROM ACCUMULATOR A TO ACCUMULATOR B

$$\underbrace{\text{TAB}}_{16}$$

Move the contents of Accumulator A to Accumulator B. Set the Sign and Zero statuses accordingly. Clear the Overflow status.

Suppose $xx = 00_{16}$. After executing the:

   TAB

instruction, Accumulators A and B will contain 0.

TAP — MOVE FROM ACCUMULATOR A TO CCR

$$\underbrace{\text{TAP}}_{06}$$

Move bits 0 - 5 in Accumulator A into the Condition Code register.

Suppose Accumulator A contains CA_{16}. After executing the:

> TAP

instruction, the CCR will be set as follows:

```
         A C I S Z O C
   CCR  |0|0|1|0|1|0|
          ↓ ↓ ↓ ↓ ↓ ↓
   CA =  1 1 0 0 1 0 1 0
```

These bits are ignored

TBA — MOVE FROM ACCUMULATOR B TO ACCUMULATOR A

Move the contents of Accumulator B to Accumulator A. Set the Sign and Zero statuses accordingly. Clear the Overflow status.

Suppose $xx = C3_{16}$. After executing the:

> TBA

instruction, Accumulators A and B will contain $C3_{16}$.

```
                    1 1 0 0 0 0 1 1 ◄——— Nonzero result, set Z to 0

    Carry is not affected

         Set S to 1 ◄—              Overflow is cleared to 0
```

6-85

TPA — MOVE CCR TO ACCUMULATOR A


```
        A  C  I  S  Z  O  C
CCR      [  ][  ][  ][  ][  ][  ]

A    [1][1][  ][  ][  ][  ][  ][  ]
B    [                          ]
INX  [                          ]
SP   [                          ]
PC   [  m m  ][  m m  ]    ( m m m m + 1 )
IR   [   0 7   ]
```

```
TPA
⎵
07
```

Move the contents of the CCR into Accumulator A, bits 0 - 5. Set Bits 6 and 7 of Accumulator A to 1.

Suppose the CCR was in the following state:

    S and C are 1.
    A_C, I, Z and O are 0.

After executing the:

    TPA

instruction, Accumulator A will contain $C9_{16}$.

```
         A  C  I  S  Z  O  C
    CCR [ 0][ 0][ 1][ 0][ 0][ 1]
          ↓  ↓  ↓  ↓  ↓  ↓
     1  1  0  0  1  0  0  1   Accumulator A
     1 ↗
```

Accumulator A is the only register affected. No statuses are altered.

TST — TEST THE CONTENTS OF ACCUMULATOR OR MEMORY

Set the Sign and Zero flags depending on the contents of the specified Accumulator or the selected memory byte.

First, consider testing an Accumulator:

Set the Sign and Zero flags depending on the result of subtracting 00_{16} from Accumulator A or B. Clear the Overflow and Carry flags.

Suppose $xx = 31_{16}$. After executing a:

    TST   B

instruction, the Sign, Zero, Overflow and Carry statuses are 0.

TST offers two memory access methods: indexed and extended.

Test the selected memory byte by subtracting 00_{16} from its contents. Set the Sign and Zero flags accordingly, and clear the Overflow and Carry statuses.

Suppose the Index register contains 0100_{16}, $cc = 02_{16}$, and the contents of memory location 0102_{16} are 00. Executing a:

    TST   2,X

instruction would set the Sign, Overflow and Carry flags to 0 and set the Zero flag to 1.

TSX — MOVE FROM STACK POINTER TO INDEX REGISTER

Move the contents of the Stack Pointer to the Index register and increment by one.

Suppose ppqq is $2AF7_{16}$. After the execution of the:

      TSX

instruction, the Index register will contain $2AF8_{16}$.

The reason the Index register is loaded with the contents of the Stack Pointer plus one is to allow the Index register to point directly at the bottom of the stack. Recall that the MC6800 employs a decrement after write, increment before read stack implementation scheme.

No other registers or statuses are affected.

TXS — MOVE FROM INDEX REGISTER TO STACK POINTER

Move the contents of the Index register to the Stack Pointer and decrement by one.

Suppose ppqq = $2AF8_{16}$. After:

      TXS

has executed, the Stack Pointer will contain $2AF7_{16}$.

No other registers or statuses are affected.

WAI — WAIT FOR INTERRUPT


```
                                                    DATA
                                                   MEMORY
                                              ┌──────────┐
                                              │11 CISZOC │ xxxx-6
                                              │   b b    │ xxxx-5
                                              │   a a    │ xxxx-4
                                              │   c c    │ xxxx-3
                                              │   d d    │ xxxx-2
          A CISZOC                            │   m m    │ xxxx-1
      CCR ┌┬┬┬┬┬┐          ┌─────────┐        │  mm + 1  │ xxxx
          └┴┴┴┴┴┘          │  c c d d │
                           └─────────┘
       A  ┌──────┐                                PROGRAM
       B  │ a a  │         ┌─────────┐            MEMORY
     INX  │ b b  │  ┌────┐ │ x x x x-7│        ┌──────────┐
      SP  │ c c  │  │d d │ └─────────┘         │          │
      PC  │ x x  │  │x x │                     │   3 E    │ mmmm
      IR  │ m m  │  │m m │ ┌──────────┐        │          │ mmmm + 1
          │ 3 E  │  └────┘ │ m m m m+1│        └──────────┘
          └──────┘         └──────────┘
```

WAI
‾‾
3E

The Program Counter is incremented by one, then the Program Counter, Index register, Accumulators A and B, and the Condition Code register are all pushed onto the stack. The registers and the corresponding memory locations into which they are pushed are shown below:

Memory Location
(SP is xxxx at start of instruction execution)   Register

Memory Location	Register
xxxx	Low byte of Program Counter
xxxx-1	High byte of Program Counter
xxxx-2	Low byte of Index register
xxxx-3	High byte of Index register
xxxx-4	Accumulator A
xxxx-5	Accumulator B
xxxx-6	Condition Code register

After the status of the system has been saved on the stack, execution is halted until a peripheral device requests an interrupt. When an interrupt is requested, the interrupt mask bit is set to 1 and a jump is made to the address contained in the normal External Interrupt Vector. Consult Chapter 6 of "An Introduction To Microcomputers: Volume II — Some Real Products" for further information on the WAI instruction.

Chapter 7
SOME COMMONLY USED SUBROUTINES

There are a number of operations which occur in many microcomputer programs, irrespective of the application. This chapter will provide a number of frequently used instruction sequences.

To make the most effective use of this chapter, you should study each subroutine until you know it well enough to modify it. As a simple exercise, you should attempt to rewrite the subroutine, so that it does the same job using fewer execution cycles, or fewer instructions, or both. Next rewrite the programs to implement variations. For example, binary multiplication of 16-bit numbers is illustrated; how about a routine to multiply 32-bit numbers? Look upon each example as a typical, illustrative instruction sequence, which you will likely modify to meet your immediate needs.

Simple programs at the level covered in this chapter fall into one of four categories:

1) Memory addressing

2) Data movement

3) Arithmetic

4) Program execution sequence logic

We will describe programs in the above category sequence.

MEMORY ADDRESSING

The MC6800 has an unusually large variety of memory referencing instructions; direct, indexed and implied (where implied addressing is a special case of indexed addressing) addressing are all available on the MC6800. Other addressing modes may be implemented through simple instruction sequences.

We are going to show auto increment, auto decrement, indirect addressing and indirect addressing with post-indexing; all of these modes are described and illustrated in "An Introduction To Microcomputers: Volume I — Basic Concepts".

AUTO INCREMENT AND AUTO DECREMENT

One of the weaknesses of the MC6800 instruction set, as compared to those of some other microcomputers, **is the lack of auto incrementing and auto decrementing implied addressing;** the data move routines described later in this chapter illustrate the gratuitous need to constantly increment/decrement addresses when handling data buffers — or any blocks of contiguous data memory bytes.

Under some circumstances, **you can use the Stack Pointer to implement implied memory addressing with auto increment or auto decrement. However, you must live with some programming restrictions:**

STACK POINTER MEMORY ADDRESSING

1) You must use the Push instruction in lieu of a write-to-memory and the Pop instruction in lieu of a read-from-memory. This restricts you to auto decrementing when writing and auto incrementing when reading.

2) The previous Stack Pointer contents address the current stack top, so it must be saved while using the Stack Pointer as a memory address register. This, of course, means you cannot use subroutines, or access the stack, until you have restored the Stack Pointer.

3) In general, it would be unwise to use the Stack Pointer in an interrupt-driven system. When an interrupt is serviced, the system status is pushed onto the stack; if the Stack Pointer is pointing into a data table at the time of the interrupt, the system status will replace a portion of the data table.

To save the Stack Pointer contents, two approaches may be used:

SAVING THE STACK POINTER

1) The Stack Pointer may be saved in memory using the

       STS   [expr]
               [expr,X]

   instruction. This instruction stores the contents of the Stack Pointer in the memory locations specified by expr or at the locations specified using the contents of the Index register and expr to form an address. This requires reserving two bytes of random access memory.

2) If the Index register is not of significance, i.e., its contents may be destroyed, the Stack Pointer may be saved in the Index register using the

       TSX

   instruction, which stores [SP] + 1 into the Index register. Note that the Index register must not be altered until the Stack Pointer contents have been restored.

Restoring the Stack Pointer contents is performed using instructions that complement the method used to save the Stack Pointer:

RESTORING THE STACK POINTER

1) If the Stack Pointer has been saved in memory, the

       LDS   [expr]
               [expr,X]

   instruction is used to restore the Stack Pointer.

2) If the Stack Pointer contents were saved in the Index register, the

       TXS

   instruction stores [IX]-1 into the Stack Pointer. Note that this instruction restores the original Stack Pointer contents when used in conjunction with the TSX instruction described above.

Once the address of the stack has been saved, the address of the memory locations to be accessed can be loaded using the LDS instruction

LOADING ADDRESS INTO STACK POINTER

       LDS   [#expr]
               [expr]
               [expr,X]

The LDS #expr form loads an immediate address into the Stack Pointer. This could be used in an environment where a buffer, e.g., a CRT input buffer, has a dedicated address. The other two forms of the LDS instruction could be used where more than one buffer or memory location is to be referenced; in this case, the address is saved in two bytes of read/write memory.

INDIRECT ADDRESSING

Indirect addressing specifies that the memory address you require is stored in two memory bytes:

In the illustration above, memory bytes 0802_{16} and 0803_{16} hold the required memory address: $0A21_{16}$.

These instructions simulate indirect addressing:

```
LDX     INDA
LDA A   0,X
```

The LDX instruction moves the address, $0A21_{16}$, into the Index register. The LDA A instruction demonstrates how to access memory location $0A21_{16}$.

INDIRECT POST-INDEXED ADDRESSING

In some applications, it is necessary or certainly preferable to perform **indirect post-indexed addressing.** Using MC6800 indexed addressing, post-indexing **can be performed in the following manner:**

```
STX     TEMP        STORE INDEX IN MEMORY
LDX     #INDA       PUT BASE ADDRESS IN INDEX REGISTER
LDA A   1,X         LOAD LOW ORDER BYTE OF INDIRECT ADDRESS
ADD A   TEMP + 1    ADD LOW ORDER BYTE OF INDEX
STA A   TEMP + 1    STORE RESULT IN MEMORY
LDA A   0,X         LOAD HIGH ORDER BYTE OF INDIRECT ADDRESS
ADC A   TEMP        ADD HIGH ORDER BYTE OF INDEX, WITH CARRY
STA A   TEMP        STORE RESULT IN MEMORY
LDX     TEMP        LOAD INDEXED INDIRECT ADDRESS INTO INDEX
                    REGISTER
```

At the beginning of this instruction sequence, we assume that the index is in the Index register. Next, the index is stored in memory so that the indirect address may be accessed via the Index register. The index is then added to the indirect address, and the result is placed in the Index register; any memory operation can now be performed using the Index register as the address. **Note that** this sequence points up a flaw in the MC6800 instruction set; that is, **the Ac-**

cumulators may not be added/stored/operated on with the Index register. Note how much simpler life would be if one could execute this sequence:

```
STX      TEMP
LDX      #INDA
LDA  A   0,X
LDA  B   1,X
LDX      TEMP
AAX
```

where the AAX instruction would be used to add the contents of Accumulators A and B to the Index register.

DATA MOVEMENT

We will now examine some instruction sequences that locate and move contiguous blocks of data bytes — data buffers of any length.

MOVING SIMPLE DATA BLOCKS

Beginning with a very simple program, consider moving the contents of a contiguous block of data memory bytes from one area of memory to another. The following memory map illustrates the data movement operation:

There are three basic approaches to this program:

1) Use the Index register to perform all necessary addressing. Since there are two addresses involved (the byte's original location and its destination), this requires saving one address when using the other. That is, the program must save the source address while filling the destination, and must save the destination address while a byte is taken from the source. This is the most general method; i.e., no matter what the system is like or where in memory the buffers are, the following method will move the data. Here is the required sequence:

	LDA B	CNT	LOAD BYTE COUNT INTO ACCUMULATOR B
	LDX	#SRCE	LOAD SOURCE ADDRESS INTO INDEX REGISTER
	STX	SRCE1	SAVE SOURCE ADDRESS IN MEMORY
	LDX	#DST	LOAD DESTINATION ADDRESS INTO INDEX REGISTER
LOOP	STX	DST1	SAVE DESTINATION ADDRESS IN MEMORY
	LDX	SRCE1	LOAD SOURCE BUFFER POINTER
	LDA A	0,X	LOAD SOURCE DATA INTO ACCUMULATOR A
	INX		INCREMENT SOURCE ADDRESS
	STX	SRCE1	SAVE INCREMENTED SOURCE ADDRESS
	LDX	DST1	LOAD DESTINATION BUFFER POINTER
	STA A	0,X	STORE SOURCE DATA INTO DESTINATION
	INX		INCREMENT DESTINATION ADDRESS
	DEC B		DECREMENT BUFFER LENGTH
	BNE	LOOP	RETURN FOR MORE IF BUFFER NOT EMPTY

Note that buffer length is limited to 256 bytes. This sequence also requires four extra RAM bytes: two for SRCE1 and two for DEST1.

2) Use the Index register for one buffer address and the Stack Pointer for the other buffer address. Recall the restriction on the use of the Stack Pointer for memory addressing; if you are in an interrupt-driven system, data can be lost if the Stack Pointer is in the middle of a table when an interrupt occurs. Here is the instruction sequence for this method:

	STS	OLDSTK	SAVE STACK POINTER
	LDS	#SRCE-1	LOAD SOURCE ADDRESS INTO STACK POINTER
	LDA B	CNT	LOAD BYTE COUNT INTO ACCUMULATOR B
	LDX	#DST	LOAD DESTINATION ADDRESS INTO INDEX REGISTER
LOOP	PUL A		INCREMENT STACK POINTER, THEN PULL SOURCE BYTE
	STA A	0,X	STORE IN DESTINATION
	INX		INCREMENT DESTINATION ADDRESS
	DEC B		DECREMENT BUFFER LENGTH
	BNE	LOOP	RETURN FOR MORE IF BUFFER NOT EMPTY
	LDS	OLDSTK	RESTORE STACK POINTER

This routine requires two RAM memory bytes for OLDSTK.

3) Use indexing to generate both addresses. This method may be used if we can guarantee that the SRCE buffer is within 256_{10} bytes of the DEST buffer. (Note that this is not the case in the example shown above.) The Index register points directly to the SRCE buffer, and, by indexing using the displacement DST-SRCE, points to the DST buffer. This instruction sequence will perform the data move:

	LDX	#SRCE	LOAD ADDRESS OF SOURCE BUFFER
	LDA B	CNT	LOAD BUFFER LENGTH
LOOP	LDA A	0,X	LOAD SOURCE BYTE
	STA A	DST-SRCE,X	STORE IN DESTINATION
	INX		INCREMENT BOTH POINTERS
	DEC B		DECREMENT BYTE COUNT
	BNE	LOOP	GO BACK FOR MORE IF NOT EMPTY

Suppose the SRCE buffer is located at 0800_{16} and the DST buffer is located at $08F0_{16}$. In this example, DST-SRCE would have the value $F0_{16}$.

MULTIPLE TABLE LOOKUPS

Next consider a multiple table lookup. This is a more complex variation of the data move which we just described.

An indefinite number of data tables have their starting addresses stored in an Index Table. The Index Table's starting address is given by the label TABX:

A number of data bytes are in temporary storage, starting at a memory location identified by the label CBASE. The actual number of data bytes can be found in a memory location identified by the label CNT. This source buffer is equivalent to the source buffer in the data move program we have just described.

The destination for the block of data is one of the Data Tables. The table number is identified by the symbol TBNO, which is loaded as immediate data. The first two bytes of every table identify the displacement to the first free byte of the table; in other words, we assume that every table is partially filled and the block of data is to be moved into the unoccupied end of the selected table. The required data movement may be illustrated as follows:

Here is the appropriate instruction sequence:

```
        STS     OLDSTK          SAVE CURRENT STACK POINTER
        LDS     #CBASE-1        LOAD SOURCE TABLE ADDRESS IN STACK POINTER
        LDA B   CNT             LOAD BUFFER LENGTH IN ACCUMULATOR B
```
THE NEXT SECTION OF CODE WILL MANEUVER THE DESTINATION ADDRESS INTO
THE INDEX REGISTER
```
        LDX     #TABX + TBNO  LOAD ADDRESS OF TARGET TABLE ADDRESS
        LDX     0,X             LOAD TARGET TABLE ADDRESS
        LDA A   0,X             LOAD NUMBER OF DISPLACEMENT BYTES (BYNO)
        STX     NDX             STORE INDEX REGISTER IN MEMORY
        ADD A   NDX + 1         ADD DISPLACEMENT TO TABLE ADDRESS
        STA A   NDX + 1         RESTORE NEW TABLE ADDRESS TO MEMORY
        BCC     DOWN            WAS THERE A CARRY?
        INC     NDX             YES. INCREMENT HIGH ORDER WORD
DOWN    LDX     NDX             LOAD DESTINATION ADDRESS
```
THE STACK POINTER HOLDS THE SOURCE ADDRESS AND THE INDEX
REGISTER HOLDS THE DESTINATION ADDRESS. NOW MOVE THE BUFFER
```
LOOP    PUL A                   LOAD SOURCE BYTE
        STA A   0,X             STORE IN DESTINATION
        INX                     UPDATE DESTINATION ADDRESS
        DEC B                   DECREMENT THE COUNT
        BNE     LOOP            RETURN FOR MORE IF NECESSARY
        LDS     OLDSTK          RESTORE STACK POINTER
```

This routine requires four extra RAM bytes: two for NDX and two for OLDSTK. Note that this process should not be used in an interrupt-driven system, as the Stack Pointer is accessing table data. Also, note that this code emphasizes a major problem with the MC6800 instruction set. There are no instructions which allow any form of data manipulation between the contents of the Index register and the contents of either Accumulator.

SORTING DATA

Both of the programming examples we have described thus far simply move a block of data from one location to another. Reorganizing data is also very important, therefore **we will illustrate a sort routine.**

The sort, as illustrated, takes a sequence of signed binary numbers, stored in contiguous memory locations and reorganizes them in ascending order, so that the smallest number comes first and the largest number comes last.

The sort routine we are going to program uses a bubble-up algorithm. Consider a sequence of numbers, where the label LIST identifies the address of the first number's storage location in memory. These are the necessary sort routine program steps:

1) Start a pass at the beginning of the LIST, initialize a flag to indicate a "no swap" condition.   `SORTING DATA`

2) Compare a consecutive pair of numbers; if the first number is smaller than the second number, do nothing; otherwise exchange the two numbers and set the flag to indicate "swap made".

3) Compare the address of the second number to the end of list address, identified by the label ENDL. If not at the end, increment so that the second number of the current pair becomes the first number of the next pair and return to step 2.

4) At the end of the list, check the "swap" flag. If any swap was made during the pass, return to step 1 to make another pass.

5) If a pass is made with no swaps, all numbers are in order. Exit.

As an example, consider the case where the numbers 1 through 10 are in reverse order. Nine ex-
changes will be made during the first-pass, at the end of which the largest number will have been
"bubbled up" to the top:

		START	AFTER 1 PASS
LIST		10	9
		9	8
		8	7
		7	6
		6	5
		5	4
		4	3
		3	2
		2	1
ENDL		1	10

Another eight passes will be needed to get all numbers in order, then a tenth pass is needed to
get a "no swap" exit condition.

SORT is implemented as a subroutine which is passed parameters in locations following the
subroutine call. Two parameters are specified.

    LIST   the beginning address of the data buffer containing numbers to be sorted

    ENDL   the ending address of the data buffer containing numbers to be sorted

Here is the sort program:

```
        JSR     SORT
        FDB     LIST
        FDB     ENDL
        -
        -
        -
SORT    TSX                     MOVE STACK POINTER TO INDEX REGISTER
        LDX     0,X             LOAD ADDRESS OF FIRST ARGUMENT
        LDX     0,X             LOAD LIST ADDRESS INTO INDEX REGISTER
        STX     TOP             STORE IN TEMPORARY RAM LOCATION
        TSX                     PUT SECOND PARAMETER
        LDX     0,X             IN INDEX REGISTER
        LDX     2,X
        STX     LAST            STORE IN TEMPORARY RAM LOCATION
LOOP1   LDX     TOP             RESTORE 'LIST' TO INDEX REGISTER
        CLR     SWITCH          CLEAR 'NO SWAP' INDICATOR
LOOP2   LDA A   0,X             LOAD ELEMENT OF LIST
        CMP A   1,X             COMPARE WITH FOLLOWING ELEMENT
        BLE     AD3             IS IT LESS THAN OR EQUAL?
        LDA B   1,X             NO — LOAD SO WE CAN SWAP
        STA B   0,X             STORE SMALLER VALUE
        STA A   1,X             STORE LARGER VALUE
        LDA B   #1              MAKE SWITCH VALUE NONZERO
        STA B   SWITCH
AD3     INX                     INCREMENT INDEX REGISTER
        CPX     LAST            COMPARE WITH ENDL ADDRESS
        BNE     LOOP2           DONE WITH THIS PASS?
        TST     SWITCH          YES. TEST 'NO SWAP' FLAG
        BNE     LOOP1           DID WE SWAP?
```

NO SWAP ON LAST PASS. PREPARE TO RETURN

```
TSX              SAVE STACK POINTER IN INDEX REGISTER
LDS    0,X       LOAD RETURN ADDRESS TO STACK POINTER
INS              INCREMENT PAST PARAMETERS TO
INS              THE NEXT INSTRUCTION
INS
INS
STS    0,X       PUT NEW RETURN ADDRESS IN STACK
TXS              RESTORE STACK POINTER
RTS              RETURN
```

ARITHMETIC

Addition, subtraction, multiplication and division will be described under this group. Transcendental functions are complex enough to require entire text books devoted to them, so we will not even broach the subject.

Even within the simple bounds of addition, subtraction, multiplication and division, there is a degree of latitude that exceeds the scope of material we can cover. Significantly different algorithms are required depending upon the magnitude of the number. Binary and decimal arithmetic also require different algorithms. Therefore, **for addition and subtraction, we will consider large or small binary or decimal numbers. For multiplication and division we consider small binary numbers only.**

BINARY ADDITION

First consider multibyte, binary addition.

Two positive, integer numbers, each CNT bytes long, are to be added. The number buffer starting addresses are given by BUF1 and BUF2. The answer is to be stored in a buffer starting at BUF3.

The multibyte addition may be illustrated as follows:

Like the data movement programs illustrated previously, there are three basic options available:

1) **The Index register does all addressing.** This is a general purpose method which occupies copious amounts of memory.

2) **The Index register and the Stack Pointer perform the required addressing.** This is a more efficient method than 1), but it should not be used in an interrupt-driven system.

3) **The Index register does** all addressing, but the buffers are arranged so that they are within 256 memory locations of each other, allowing **indexed addressing with displacement.** This is the preferred method. The requisite instruction sequence is presented below:

```
        LDX     #BUFA        LOAD INDEX REGISTER WITH BUFFER ADDRESS
        LDA B   CNT          LOAD BUFFER LENGTH INTO ACCUMULATOR B
        CLC                  CLEAR CARRY
LOOP    LDA A   0,X          LOAD NEXT BUFA BYTE
        ADC A   BUFB-BUFA,X  ADD NEXT BUFB BYTE
        STA A   BUFC-BUFA,X  SAVE IN NEXT ANSWER BUFFER BYTE
        INX                  INCREMENT BUFFER ADDRESS
        DEC B                DECREMENT COUNTER
        BNE     LOOP         RETURN FOR MORE BYTES
```

BINARY SUBTRACTION

Because the MC6800 has special subtraction instructions, binary subtraction is almost identical to binary addition. In either subroutine, simply replace the ADC instruction with the SBC instruction and accurate binary subtraction will result.

DECIMAL ADDITION

The presence of a DAA instruction makes decimal addition very easy using the MC6800 microcomputer. **Simply insert a DAA instruction to follow the ADC** in any binary addition program and you have decimal addition.

```
       -
       -
       -
LOOP   LDA A   0,X           LOAD NEXT BUFA BYTE
       ADC A   BUFB-BUFA,X   ADD NEXT BUFB BYTE
       DAA                   DECIMAL ADJUST RESULT
       STA A   BUFC-BUFA,X   SAVE IN ANSWER BUFFER
```

One cautionary note, however: the decimal addition routine created by including a DAA instruction in the binary addition routine assumes that valid binary-coded-decimal (BCD) **data is stored in the source buffers.** If, by mistake, you have invalid data in either source buffer, you will generate a meaningless answer — and not know it.

If your program is one which cannot guarantee that data in source buffers is valid binary-coded-decimal, then you must write a routine to check buffer contents and ensure that no high or low 4-bit unit within any byte contains a binary code of A through F.

DECIMAL SUBTRACTION

Decimal subtraction is complicated somewhat by the fact that you cannot use the MC6800 subtract instructions; these instructions only work for binary data, since they automatically generate the twos complement of the subtrahend. As described in "An Introduction To Microcomputers", Volume I, binary-coded-decimal subtraction requires that you take the tens complement of the subtrahend.

Let us return to the binary addition program and create, in its place, a decimal subtraction equivalent; here is the appropriate memory map:

DATA MEMORY

MINU ◄───── Low order digit
MINU + 1
MINU + 2

SBTRA ◄───── Low order digit
SBTRA + 1
SBTRA + 2

RSLT ◄───── Low order digit
RSLT + 1
RSLT + 2

CNT ◄───── Buffer length stored here

Here is the required instruction sequence:

```
        LDX     #MINU       LOAD ADDRESS OF MINUEND BUFFER
        LDA  B  CNT         LOAD BUFFER LENGTH INTO ACCUMULATOR B
        LDA  A  #$80        SET (CARRY) INDICATING NO BORROW
LOOP    ROL  A              RESTORE CARRY FROM ACCUMULATOR A
        LDA  A  #$99        LOAD $99 INTO ACCUMULATOR A
        ADC  A  0           ADD ZERO WITH CARRY
        SUB  A  SBTRA-MINU,X PRODUCE NINES COMPLEMENT OF SUBTRAHEND
        ADD  A  0,X         ADD MINUEND
        DAA                 DECIMAL ADJUST RESULT
        STA  A  RSLT-MINU,X STORE RESULT
        ROR  A              SAVE CARRY FROM DECIMAL ADJUST
        INX                 INCREMENT ADDRESS
        DEC  B              DECREMENT BYTE COUNT
        BNE     LOOP        GO BACK FOR NEXT TWO DIGITS
```

MULTIPLICATION AND DIVISION

Multiplication and division must be approached with an element of caution within microcomputer systems. These are operations which are unsuited to the organization of a microcomputer; any nontrivial multiplication or division can take so long to execute that it will severely degrade overall performance. **If your microcomputer application is going to make extensive use of multiplication, division or transcendental functions, you should seriously consider using one of the many calculator/arithmetic chips that are now commercially available.** Transferring complex arithmetic to such a chip can make the difference between a microcomputer system being viable or nonviable in your application.

You can implement simple multiplication and division in microcomputer systems that do not make extensive, or time-consuming use of these routines; therefore we will describe some simple program sequences.

8-BIT BINARY MULTIPLICATION

Consider the multiplication of two unsigned, 8-bit data values, to generate a 16-bit product. The simplest way of performing this multiplication is to add the multiplier to 0 the number of times given by the multiplicand. For example, **you can multiply 4 by 3 if you add 4 to 0 three times.**

Suppose memory location MULT contains the multiplicand and memory location ARG contains the multiplier. The following routine performs the operation, returning the 16-bit result in Accumulator A (high order) and B (low order):

```
        CLR  A              CLEAR ACCUMULATORS A AND B
        CLR  B              TO INITIALIZE RESULT
        TST    ARG          TEST FOR 0 IN ARG
        BEQ    LEAVE        RETURN IF ZERO
NEXT    ADD B  MULT         ADD MULTIPLICAND TO LOW ORDER BYTE
        BCC    DOWN         DID WE GET A CARRY?
        INC  A              YES. INCREMENT HIGH ORDER BYTE
DOWN    DEC    ARG          DECREMENT MULTIPLIER
        BNE    NEXT         ADD AGAIN IF NOT FINISHED
LEAVE   RTS                 RETURN WHEN MULTIPLIER IS ZERO
```

This routine could be a very fast one (if ARG is 0, only five instructions will execute) or a very slow one — if ARG is 255, then this routine could take up to 1280 instruction executions.

In general, there is a faster way of executing multiplications. We can use the fact that a binary digit is limited to having values of 0 or 1; this means that at the single digit level, multiplication degenerates to addition or no addition.

Let us explain this concept; using common decimal notation, consider the following multiplication:

```
                    1 4 2       Multiplicand
                  x 3 0 7       Multiplier
                  4 2 6 0 0
                    0 0 0 0     Partial Product
                      9 9 4
                  4 3 5 9 4     Product
```
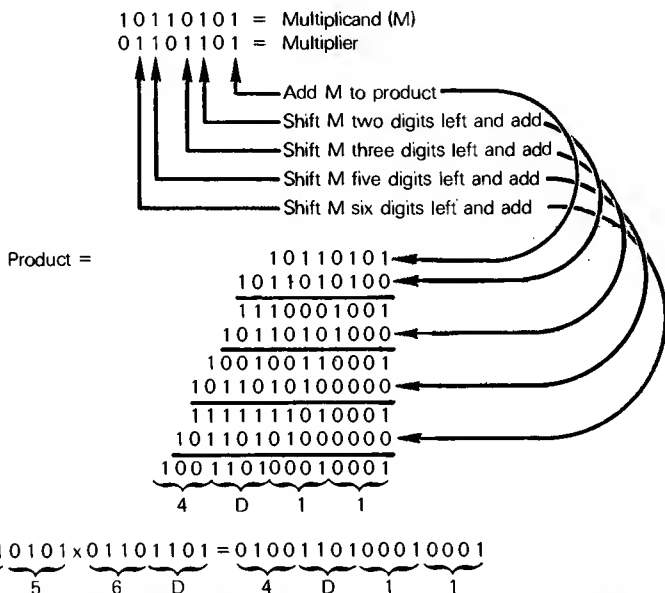
```
  142  =  Multiplier
  307  =  Multiplicand
```

Add 7 x Multiplier to product

Shift Multiplier two digits left, then multiply by 3 and add to product

Each partial product equals the multiplicand being multiplied by one digit of the multiplier. The partial product is shifted to the left by tacking on 0s to the right. The number of 0s tacked on to the right is equal to the number of digits to the right of the current multiplier digit:

We can extend this same concept to binary arithmetic, in which case the problem becomes very simple, since no binary digit can have a value other than 0 or 1. This being the case, you have only two choices: wherever a multiplier digit is 0, you do not add the shifted multiplicand to the answer; but if the multiplier digit is 1 you do add the shifted multiplicand to the answer. Here is an example:

Using the "shift-and-add" technique, the following steps will multiply a one-byte multiplicand by a one-byte multiplier to produce the correct two-byte result:

a) Test the least significant bit of the multiplier. If zero, go to Step b. If one, add the multiplicand to the most significant byte of the result.

b) Shift the entire two-byte result right one bit position.

c) Repeat Steps a and b until all 8 bits of the multiplier have been tested.

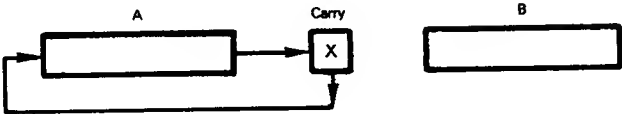Consider B5 * 6D, the binary multiplication we just illustrated:

Multiplier = 0 1 1 0 1 1 0 1
Multiplicand = 1 0 1 1 0 1 0 1

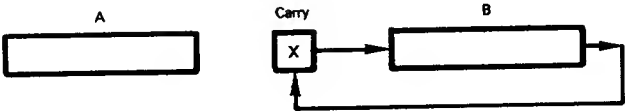|		RESULT	
		HIGH ORDER BYTE	LOW ORDER BYTE
	Start:	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
0 1 1 0 1 1 0 **1**	Step 1 (a)	1 0 1 1 0 1 0 1	0 0 0 0 0 0 0 0
	1 (b)	0 1 0 1 1 0 1 0	1 0 0 0 0 0 0 0
0 1 1 0 1 1 **0** 1	Step 2 (a,b)	0 0 1 0 1 1 0 1	0 1 0 0 0 0 0 0
0 1 1 0 1 **1** 0 1	Step 3 (a)	1 0 1 1 0 1 0 1	
		1 1 1 0 0 0 1 0	0 1 0 0 0 0 0 0
	3 (b)	0 1 1 1 0 0 0 1	0 0 1 0 0 0 0 0
0 1 1 0 **1** 1 0 1	Step 4 (a)	1 0 1 1 0 1 0 1	
		c→1 0 0 1 0 0 1 1 0	0 0 1 0 0 0 0 0
	4 (b)	1 0 0 1 0 0 1 1	0 0 0 1 0 0 0 0
0 1 1 **0** 1 1 0 1	Step 5 (a,b)	0 1 0 0 1 0 0 1	1 0 0 0 1 0 0 0
0 1 **1** 0 1 1 0 1	Step 6 (a)	1 0 1 1 0 1 0 1	
		1 1 1 1 1 1 1 0	1 0 0 0 1 0 0 0
	6 (b)	0 1 1 1 1 1 1 1	0 1 0 0 0 1 0 0
0 **1** 1 0 1 1 0 1	Step 7 (a)	1 0 1 1 0 1 0 1	
		c→1 0 0 1 1 0 1 0 0	0 1 0 0 0 1 0 0
	7 (b)	1 0 0 1 1 0 1 0	0 0 1 0 0 0 1 0
0 1 1 0 1 1 0 1	Step 8 (a,b)	0 1 0 0 1 1 0 1	0 0 0 1 0 0 0 1
		4 D	1 1

We will now write a program to implement this multiplication algorithm.

The 16-bit right shift of the result is performed by two rotate-right-through-carry instructions as follows:

Rotate Accumulator A:

Then rotate Accumulator B to complete the shift:

As in the previous example, memory location MULT contains the multiplicand and memory location ARG contains the multiplier. The following routine will perform the operation, returning the 16-bit result in Accumulators A (high order) and B (low order):

```
        LDA  A   #8          INITIALIZE BIT COUNT
        STA  A   CNT
        CLR  A               CLEAR HIGH ORDER RESULT REGISTER
MULT1   ASR      ARG         ROTATE LEAST SIGNIFICANT BIT OF
        BCC      MULT2       MULTIPLIER TO CARRY AND TEST
        ADD  A   MULT        BIT IS 1. ADD TO HIGH ORDER BYTE
MULT2   ROR  A               ROTATE HIGH ORDER BYTE
        ROR  B               ROTATE LOW ORDER BYTE
        DEC      CNT         DECREMENT BIT COUNT
        BNE      MULT1       REPEAT IF NOT FINISHED
        RTS                  RETURN WHEN BIT COUNT IS ZERO
```

8-BIT BINARY DIVISION

An analogous procedure is used to divide an unsigned 16-bit number by an unsigned 8-bit number. Here, **the process involves subtraction rather than addition, and rotate-left instructions instead of rotate-right instructions.**

For the program below, the dividend is in memory locations DIVD (high order byte) and DIVD + 1. The divisor is in memory location DIVS. The 8-bit quotient is returned in Accumulator B, and the remainder is returned in Accumulator A.

```
DIV      LDA  A   #8          INITIALIZE BIT COUNT
         STA  A   CNT
         LDA  A   DIVD        LOAD DIVIDEND INTO
         LDA  B   DIVD + 1    ACCUMULATORS A AND B
         ASL  B               SHIFT MSB OF LOW BYTE INTO CARRY
ITERATE  ROL  A               ROTATE CARRY INTO LSB OF REMAINING DIVIDEND
         SUB  A   DIVS        SUBTRACT DIVISOR. IF LESS THAN HIGH BYTE
         BCC      NEXT        OF REMAINING DIVIDEND, GO TO NEXT
         ADD  A   DIVS        OTHERWISE, ADD IT BACK AND SET CARRY
NEXT     ROL  B               ROTATE CARRY TO QUOTIENT; MSB TO CARRY
         DEC      CNT         DECREMENT BIT COUNT
         BNE      ITERATE     ITERATE LOOP IF NOT ZERO
         COM  B               COMPLEMENT THE QUOTIENT
         RTS                  LEAVE THIS ROUTINE
```

16-BIT BINARY MULTIPLICATION

Now consider the multiplication of two 16-bit numbers, yielding a 32-bit result.

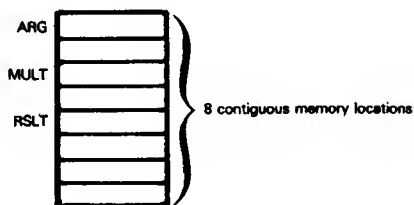**The algorithm used is a simple extension of 8-bit multiplication:**

1) Shift the multiplier right into the Carry.

2) If Carry is 0 go to Step 4.

3) Add the multiplicand to the two high bytes of the result.

4) Shift the result right one bit.

5) Done? If not, go to Step 1.

In this case, MULT is the 16-bit multiplicand, ARG is the 16-bit multiplier and **this is the required instruction sequence:**

```
         CLR     RSLT        CLEAR FOUR MEMORY LOCATIONS TO
         CLR     RSLT + 1    HOLD THE RESULT
         CLR     RSLT + 2
         CLR     RSLT + 3
         LDA A   #16         INITIALIZE THE COUNTER
         STA A   CNT
UP       ROR     ARG         ROTATE LSB OF MULTIPLIER INTO
         ROR     ARG + 1     THE CARRY
         BCC     NEXT        WAS CARRY SET TO 1?
         LDA A   MULT + 1    YES. ADD MULTIPLICAND
         ADD A   RSLT + 1    ADD LOW ORDER BYTE OF MULTIPLICAND
         STA A   RSLT + 1    TO THE RESULT
         LDA A   MULT        ADD THE HIGH ORDER BYTE OF
         ADC A   RSLT        THE MULTIPLICAND
         STA A   RSLT
NEXT     ROR     RSLT        ROTATE THE RESULT
         ROR     RSLT + 1    ONE BIT TO THE RIGHT
         ROR     RSLT + 2
         ROR     RSLT + 3
         DEC     CNT         DECREMENT THE COUNT
         BNE     UP          REPEAT IF NOT FINISHED
```

Note that almost all the instructions in this sequence are three-byte memory reference instructions. **The number of bytes of object code used for this routine can be shortened considerably if** locations ARG, MULT and RSLT are contiguous. In this case, the Index register can be made to point to the first location, and **all references to memory can be made through the Index register,** thereby saving 15 bytes of code. Consider the following example:

8 contiguous memory locations

This would be the necessary sequence:

```
          LDX     #ARG      LOAD INDEX REGISTER WITH TABLE ADDRESS
          CLR     4,X       CLEAR RESULT LOCATION
          CLR     5,X
          CLR     6,X
          CLR     7,X
          LDA A   #16       INITIALIZE COUNT
          STA A   CNT
UP        ROR     0,X       ROTATE MULTIPLIER RIGHT INTO CARRY
          ROR     1,X
          BCC     NEXT      CARRY SET?
          LDA A   3,X       YES — ADD MULTIPLICAND TO RESULT
          ADD A   5,X
          STA A   5,X
          LDA A   2,X
          ADC A   4,X
          STA A   4,X
NEXT      ROR     4,X       ROTATE RESULT RIGHT
          ROR     5,X
          ROR     6,X
          ROR     7,X
          DEC     CNT       DECREMENT COUNT
          BNE     UP        REPEAT IF COUNT NOT ZERO
```

Note that it might be possible to load the data onto the stack. In this case, instead of executing a LDX #ARG instruction, a TSX instruction would point the Index register at the top of the stack. However, this method is slower than the first method; direct or extended memory reference is always faster than indexed memory addressing.
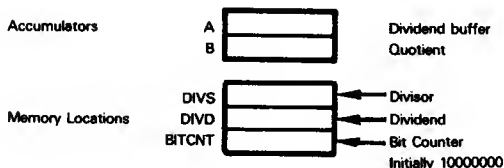
BINARY DIVISION

Consider simple 8-bit division. $B3_{16}$ divided by 15_{16} may be illustrated as follows:

```
                                      1000 ◄──── Quotient
          Divisor ──────► 10101 )10110011 ◄──── Dividend
                                10101
                                ─────
                                 1011
```

The result is 8_{16} with a remainder of B_{16}.

The division algorithm works by shifting the dividend into a register that is initially cleared. Whenever the dividend shift buffer contents exceed the divisor, the divisor is subtracted from the shift buffer contents and a binary 1 digit is inserted into the appropriate quotient bit position.

Consider the following register and memory assignments:

| Accumulators | A | | Dividend buffer |
| | B | | Quotient |

Memory Locations	DIVS		◄── Divisor
	DIVD		◄── Dividend
	BITCNT		◄── Bit Counter
			Initially 10000000

Initially, DIVD holds the dividend and DIVS holds the divisor. The quotient will be generated in Accumulator B; the remainder will be left in Accumulator A. This is the division program which results:

```
          LDA  A    #$80        INITIALIZE BIT COUNTER
          STA  A    BITCNT
          CLR  A                CLEAR ACCUMULATORS A AND B AND
          CLR  B                THE CARRY BIT
LOOP      ROL       DIVD        SHIFT DIVD AND ACCUMULATOR A
          ROL  A                AS A 16-BIT UNIT
          CMP  A    DIVS        COMPARE DIVISOR WITH DIVIDEND BUFFER
          BLT       NEXT        IS DIVISOR SMALLER?
          SUB  A    DIVS        YES. SUBTRACT DIVISOR AND OR
          ORA  B    BITCNT      THE CORRECT BIT INTO QUOTIENT REGISTER
NEXT      LSR       BITCNT      NO. SHIFT BITCNT RIGHT
          BCC                   IF CARRY NOT SET, RETURN FOR NEXT BIT
```

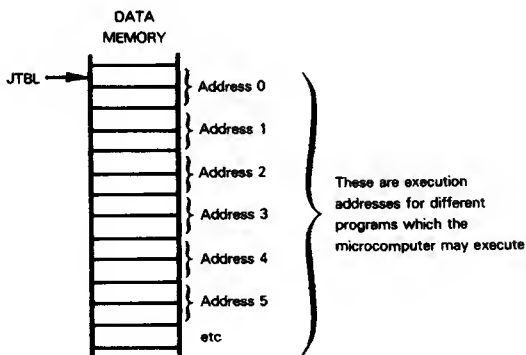# PROGRAM EXECUTION SEQUENCE LOGIC

THE JUMP TABLE

There is really only one program sequence that needs to be described under this heading: it is the Jump Table.

Remember that the MC6800 instruction set is rich in conditional instructions. The Branch instruction has fourteen conditional variations, which means that special routines are not required when your logic can go one of two ways only.

When you have three or more options, the Jump Table becomes an effective programming tool.

At the heart of a Jump Table there will be a sequence of 16-bit addresses stored in pairs of contiguous memory bytes:

We will presume that these contiguous memory addresses represent the starting addresses for a number of different programs. Assuming that the required program is identified by a program number in Accumulator A, **the following instruction sequence causes execution to**

branch to the program whose number is stored in Accumulator A:

```
        LDX       #JTBL       LOAD TABLE BASE ADDRESS IN MEMORY
        STX       TEMP
        ASL  A                MULTIPLY ACCUMULATOR A BY TWO
        ADD  A    TEMP + 1    ADD TO LOW ORDER ADDRESS BYTE
        STA  A    TEMP + 1
        LDA  A    TEMP        ADD CARRY, IF ANY, TO HIGH ORDER
        ADC  A    #0          ADDRESS BYTE
        STA  A    TEMP
        LDX       TEMP        INDEX REGISTER ADDRESSES REQUIRED ADDRESS
        LDX       0,X         LOAD REQUIRED ADDRESS IN INDEX REGISTER
        JMP       0,X         JUMP TO START OF PROGRAM
TEMP    RMB       2           RESERVE TWO BYTES FOR 'TEMP'
```

NOTES

NOTES

ABOUT THE AUTHORS OF THIS BOOK

Adam Osborne is president of Osborne and Associates, Inc., a California corporation.

Osborne and Associates, Inc., are microcomputer consultants. We will design your microcomputer based product for you, or we will help you do the job for yourself. We also deliver custom, in-house seminars on microcomputers, their future potential or their immediate use.

For manufacturers in the microcomputer and minicomputer industries, Osborne and Associates prepare technical manuals.

To order additional copies of this book, or to inquire about our services, write or telephone:

Osborne and Associates, Inc.
P.O. Box 2036
Berkeley, California 94702
(415) 548-2805

OTHER BOOKS IN THIS SERIES